

---

COMPUTE!'s

---

# 1 BOOK OF ATARI 1ST

Quality games, educational programs, tutorials, and other information for users of the Atari ST. Includes sections on programming in BASIC, Pascal, and C.

A **COMPUTE! Books** Publication

\$16.95







# COMPUTE!'s FIRST BOOK OF THE ATARI ST

**COMPUTE!**™ Publications, Inc. 

Part of ABC Consumer Magazines, Inc.  
One of the ABC Publishing Companies

Greensboro, North Carolina



The following programs were originally published in *COMPUTE!* magazine, copyright 1986, COMPUTE! Publications, Inc.

"Doodler" (February); "Switchbox" (March); "Adding System Power to ST BASIC" (Part 1, April; Part 2, May); "Hickory, Dickory, Dock" (May); "Custom Title Bars for ST BASIC" (June); "ST Hints and Tips" (June); "ST System Software, Inside Out" (June); "Odd Facets of GEM" (July); "GEM Quirks" (August); "MODified Shapes for Atari ST" (August); "Softball Statistics" (August); "3-D Tic-Tac-Toe" (September); "Home Financial Calculator" (September); "Reversi" (October).

Copyright 1986, COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-020-3

The authors and publisher have made every effort in the preparation of this book to insure the accuracy of the programs and information. However, the information and programs in this book are sold without warranty, either express or implied. Neither the authors nor COMPUTE! Publications, Inc., will be liable for any damages caused or alleged to be caused directly, indirectly, incidentally, or consequentially by the programs or information in this book.

The opinions expressed in this book are solely those of the authors and are not necessarily those of COMPUTE! Publications, Inc.

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Atari, Atari 520ST, Atari 1040ST, *NEOchrome*, ST, and TOS are trademarks or registered trademarks of Atari Corporation. GEM is a trademark of Digital Research, Inc. *Personal Pascal* is a trademark of Optimized Systems Software. *Turbo Pascal* is a trademark of Borland International, Inc.



# Contents

<i>Foreword</i> .....	<i>v</i>
<b>1. Getting Started</b> .....	<b>1</b>
Introduction to the ST	
Tom R. Halfhill .....	3
ST System Software, Inside Out	
Bill Wilkinson .....	26
Odd Facets of GEM	
Bill Wilkinson .....	29
GEM Quirks	
Bill Wilkinson .....	32
<b>2. Games</b> .....	<b>35</b>
Switchbox	
Todd Heimarck / Version by Kevin Mykytyn .....	37
Reversi	
Kevin Mykytyn / Article by Philip I. Nelson .....	47
3-D Tic-Tac-Toe	
David Bohlke .....	54
<b>3. Applications and Education</b> .....	<b>59</b>
Hickory, Dickory, Dock	
Barbara H. Schulak	
Version by Kevin Mykytyn and David Florance .....	61
Multiple-Choice Test Generator	
C. Regena .....	66
Memory Trainer	
C. Regena .....	76
Softball Statistics	
Roger Felton / Version by George Miller .....	80
Home Financial Calculator	
Patrick Parrish / Version by George Miller .....	96
<b>4. BASIC Programming</b> .....	<b>117</b>
ST Hints and Tips	
George Miller .....	119
ST BASIC Sorting Algorithms	
C. Regena .....	126
Custom Title Bars for ST BASIC	
George Miller .....	133



Adding System Power to ST BASIC	
Kevin Mykytyn .....	135
File Handling in ST BASIC	
Tony Roberts .....	150
Using GEMSYS and VDISYS in ST BASIC	
Philip I. Nelson .....	167
<b>5. Sound and Graphics .....</b>	<b>185</b>
ST Graphics	
Brian Flynn .....	187
MODified Shapes for Atari ST	
Robert G. Geiger .....	203
NEOchrome: The Rainbow Machine	
Advanced Color Features of NEOchrome	
Lee Noel, Jr., and Selby Bateman .....	213
Doodler	
D. W. Neuendorf .....	226
Making Music on the ST	
David Florance .....	232
<b>6. C Programming .....</b>	<b>247</b>
Introduction to C Programming	
Sheldon Leemon .....	249
Moving Objects in C	
Charles Brannon .....	261
<b>7. Pascal Programming .....</b>	<b>279</b>
A First Look at Pascal Programming	
Tony Roberts .....	281
Event Management and Windows in Pascal	
Program by Mark Rose / Text by Bill Wilkinson ....	290
<b>Appendices</b>	
A. A Beginner's Guide to Typing In Programs .....	323
B. Using the <i>First Book of Atari ST</i> Disk .....	325
Index .....	327
Disk Coupon .....	329



# Foreword

*COMPUTE!'s First Book of Atari ST* is an instant resource of easy-to-use programs and information that will make your ST even more valuable. There's much in these 27 articles for every age group and every interest, and many of these articles are published here for the first time. Children will enjoy learning to tell time with "Hickory, Dickory, Dock." Game players will match wits with "Reversi" and "3-D Tic-Tac-Toe." If you're a softball fan, you can track your favorite team's progress with "Softball Statistics." Or, if your memory needs improvement, "Memory Trainer" will give you a good workout.

For programmers, there are utilities, demonstrations, hints and tips, plus exciting application programs that will enhance the value of your ST as well as expand your own knowledge and techniques. Just for starters, you'll learn more about the Graphics Environment Manager (GEM) and ST system software; you'll find seven different sorting subroutines that can be incorporated into your own programs; and you'll gain insight into ST BASIC file handling.

If you want to know more about sound and graphics, *COMPUTE!'s First Book of Atari ST* provides you with a music-generating program, a drawing program written in Logo, and a detailed guide to creating animated figures with *NEOchrome*.

When ST BASIC won't accomplish what you're after, Pascal or C may be the answer. Both of these languages are faster than BASIC and give you full access to the power of GEM. You'll get a taste of how they work and how they're best used.

Each program in *COMPUTE!'s First Book of Atari ST* has been carefully tested and is ready to type in. All you need are this book and your ST, and you'll be off and running.

All the programs in *COMPUTE!'s First Book of Atari ST* are ready to type in and run. If you prefer not to type in the programs, however, you can purchase a disk that includes all the programs in the book. Call toll-free 1-800-346-6767; in New York call 1-212-887-8525. Or use the coupon found in the back of this book.





# CHAPTER ONE

---

# Getting Started





# Introduction to the ST

Tom R. Halfhill

---

*"We aren't selling home computers. We aren't selling business computers. We're selling personal computers. People can use them for whatever they want." With those words, Jack Tramiel announced the ST series in January 1985 and launched a new beginning for the Atari Corporation. Here's an introduction to the 520ST and 1040ST—the most powerful Ataris ever.*

The old stereotypes about home computers are being challenged. Now there's a new generation of machines that can match the speed and power of the most popular business-oriented personal computers installed in executive offices. They combine such features as massive memory, high-speed processing, fast floppy disk drives, hard disk interfaces, stunning graphics, and sophisticated sound into a single, reasonably priced package. They are true general-purpose personal computers—powerful enough to run state-of-the-art business software, yet versatile enough to excel at running home entertainment and educational programs.

The Atari 520ST was the first of this new breed. Announced at the Winter Consumer Electronics Show in January 1985, it quickly attracted a loyal following that is growing by the thousands each month. In January 1986, it was followed by the announcement of the 1040ST, an even more powerful computer. Here's a rundown of their features:

- The 520ST has 512K of random access memory (RAM), half a megabyte. The 1040ST has 1024K of RAM, a full megabyte. The 520ST can be expanded to a megabyte with the addition of a memory board available from several independent manufacturers.
- Both computers are built around the Motorola 68000 microprocessor. This 16/32-bit chip is clocked at 8 megahertz and can directly address up to 16 megabytes of memory with-

## CHAPTER ONE

---

out bank-switching. It's the same central processing unit found in the Apple Macintosh and Commodore Amiga, except the clock speed is slightly higher.

- High-speed microfloppy disk drives. Although the floppy interface is serial, not parallel, it transfers data at a megabit per second. Two different drives are available from Atari: a single-sided drive that stores about 360K on a 3½-inch disk and a double-sided drive that stores about 720K per disk. The 520ST can be hooked up to one or more external drives. The 1040ST comes with a built-in double-sided drive and can be hooked up to additional external drives.

- One of the fastest hard disk interfaces in personal computing. This built-in interface transfers data at an incredible 1.33 megabytes per second, faster than some RAM disks on other computers. Hard disks with capacities of 10 or 20 megabytes and more are available from Atari and other manufacturers. Prices for 20-megabyte drives are typically around \$700.

- Built-in Centronics-standard parallel port and RS-232 serial port for printers, modems, and other peripherals. These ports are compatible with IBM cables for printers and modems.

- Built-in Musical Instrument Digital Interface (MIDI) for attaching keyboard synthesizers, sequencers, drum boxes, and other electronic musical devices. Both MIDI IN and MIDI OUT ports are standard. Because the MIDI ports transfer data at a very high speed (31.25 kilobaud), they've also been considered for such future applications as inexpensive local area networks (LANs).

- A slot for cartridges containing up to 128K of read only memory (ROM).

- Intelligent audio/video port that recognizes whether an RGB (red-green-blue) color or monochrome monitor is plugged into the computer, allowing the operating system to adjust itself accordingly. This port has pins for audio input/output, and 520STs made after early 1986 also have composite video output.

- RF (radio frequency) output for hooking up to ordinary TV sets. This feature is present only on 520STs made after early 1986.

- Analog RGB color monitor (optional). One of the lowest-priced analog RGB monitors in the industry, it's capable of sharply resolving the ST's advanced graphics. Two screen



modes are available in color: 320 × 200 pixels with 16 simultaneous colors, or 640 × 200 with 4 simultaneous colors. In either mode, the color palette can be chosen from a selection of 512 possible colors.

- High-resolution monochrome monitor (optional). In monochrome mode, the ST offers the sharpest, highest-resolution screen available as a standard feature on any personal computer. With a resolution of 640 × 400 pixels and a screen refresh rate of 70 hertz—about 16 percent faster than normal monitors and TVs—this mode is unusually sharp and steady.

- Three-channel General Instruments sound chip. Envelope registers allow the chip to simulate various types of musical instruments and sound effects.

- A ROM-based operating system called TOS (Tramiel Operating System), which combines Digital Research's CP/M-68K and GEM (Graphics Environment Manager). CP/M-68K is the 68000 version of the popular Z80-based operating system CP/M (Control Program for Microcomputers), similar to MS-DOS on the IBM PC. CP/M-68K is vastly expanded, however, with provisions to support up to 16 disk drives with 512 megabytes per drive and 32 megabytes per file. To make this operating system easier to use, it is linked on the ST with GEM, a graphics-oriented user interface with icons, windows, and drop-down menus. The GEM desktop is manipulated with a mouse controller that comes with the ST. The two-button mouse plugs into one of the two controller ports built into the computer.

- Digital Research Logo and ST BASIC programming languages are included on a language disk supplied with the 520ST and 1040ST. Also included on disk are *NEOchrome*, a graphics-design program, and *1ST Word*, a word processor.

- Both STs have an 84-key keyboard with cursor keypad, numeric keypad, and ten special function keys.

- The 1040ST has built-in power supplies for the computer and internal disk drive. The 520ST uses external power-supply transformers.

### The GEM Desktop

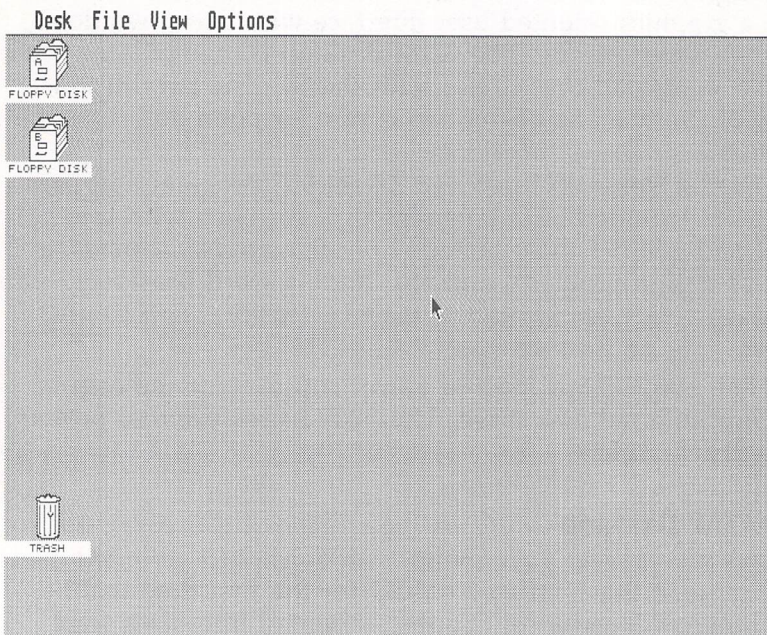
If you've never used a computer with a graphics-oriented user interface, working with an Atari ST for the first time will be an unfamiliar experience. Graphics-oriented interfaces were

pioneered by Xerox in the 1960s on experimental computers, but were popularized by the introduction of the Macintosh in 1984. Since then, they've spread to almost all personal computers and seem to be the wave of the future. You can find GEM on the Atari ST and IBM PC, the Intuition Workbench on the Commodore Amiga, *Desqview*, *Topview*, and *Microsoft Windows* on the PC and AT, and even *GEOS* on the Commodore 64.

Most of these user interfaces employ pull-down or drop-down menus, icons, multiple screen windows, and mouse controllers. Their goal is not only to reduce the number of cryptic commands that must be memorized and entered by users, but also to encourage or enforce uniformity among the wide variety of application programs. The idea is that once you learn how to use one application, you can adapt quickly to others, because they all use similar menus and screen displays for common tasks (such as loading and saving files).

The central feature of this style of "working environment" is a screen called the *desktop*. On the ST with GEM, this is the first screen you see after switching on the computer. Earlier

**Figure 1. The GEM Desktop**



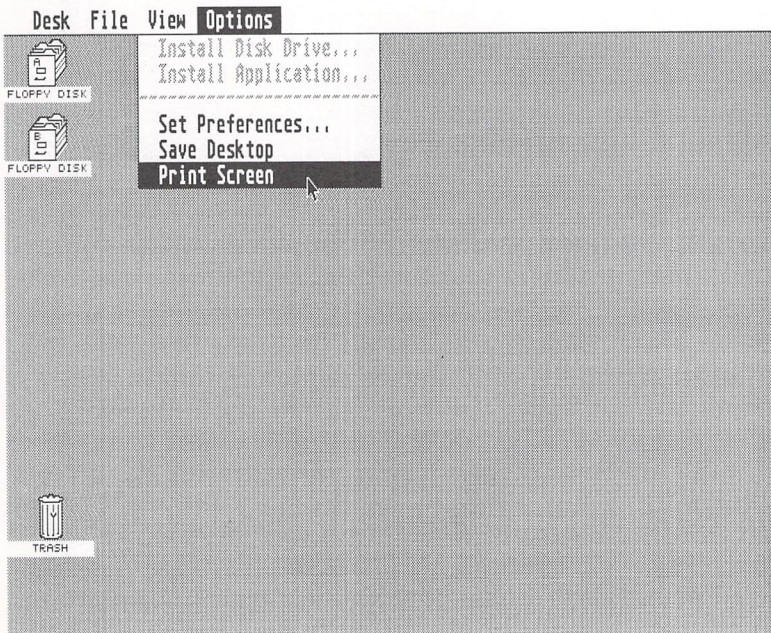


computers landed you either in BASIC or some type of disk operating system (DOS). But the ST doesn't wake up with a READY prompt, command line, or DOS menu. Instead, the first thing you see is the GEM desktop.

Icons along the edges of the desktop screen represent file drawers and a trash can (Figure 1). The drawers represent disk drives—floppy disks, hard disks, or RAM disks, depending on your system configuration. Menu titles appear across the top of the screen. Floating above the desktop is an arrow cursor that you can move by rolling the mouse or by pressing certain keys. It represents an extension of your hand on the screen.

To view a menu, you move the pointer to the desired title. Instantly, the menu drops down over the screen. (GEM's *drop-down* menus are summoned in a slightly different way than are the *pull-down* menus more often used by other systems: You don't have to click and hold the mouse button.) As you move the pointer up and down the menu, it highlights various options. Some options may be invalid for a particular operation; they appear in dim print (ghosted) and cannot be highlighted. To select an option, you simply highlight it and click the left button on the mouse (Figure 2).

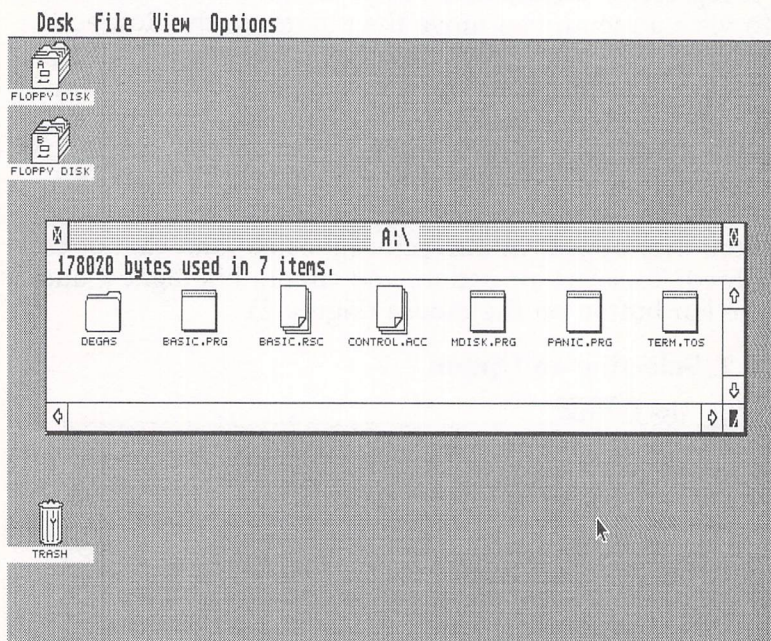
**Figure 2. Selecting an Option**





To call a disk directory, you move the pointer atop the appropriate file drawer icon and perform what's called a *double-click*—you press the mouse button twice in rapid succession. In seconds, a window appears on the desktop. (If the file drawer icon changes color, but no window opens, you didn't double-click fast enough; try again.) Various types of icons inside the window denote executable program files, data files, and subdirectories on the disk (Figure 3). We'll explain what these icons mean in a moment.

**Figure 3. Icons**



If you prefer a more conventional disk directory, you can drop down the View menu and select Show As Text. The file icons change into a list of filenames, which includes such information as file lengths in bytes and the dates on which the files were last updated. Other options on the View menu let you sort the directory by filename (alphabetically), file type, size, or date.

If you're working with a two-drive system, you can call the directory for drive B by double-clicking on its icon. When this window appears, it may overlap the window for drive A.

But the drive A window isn't erased; by pointing to it and clicking the left mouse button once, you can move it atop the drive B window. A similar click on the drive B window brings it to the fore.

You can flip back and forth between several windows in this manner, much like shuffling papers on a real desktop. Options selected from menus, such as View As Text, always affect the *active window*—the window currently on top of the pile.

### Three Kinds of Icons

All other functions of the GEM desktop work in similar ways: You point to a menu option or icon, then click the left mouse button once or twice.

As mentioned above, three types of icons can appear inside the windows of opened file drawers. An *executable program file* is any kind of application program that can be loaded and run—a word processor, a database manager, a terminal program, a game, a programming language interpreter or compiler, or whatever. Icons for executable program files are small squares with a dark band across the top, representing a screen and menu title bar.

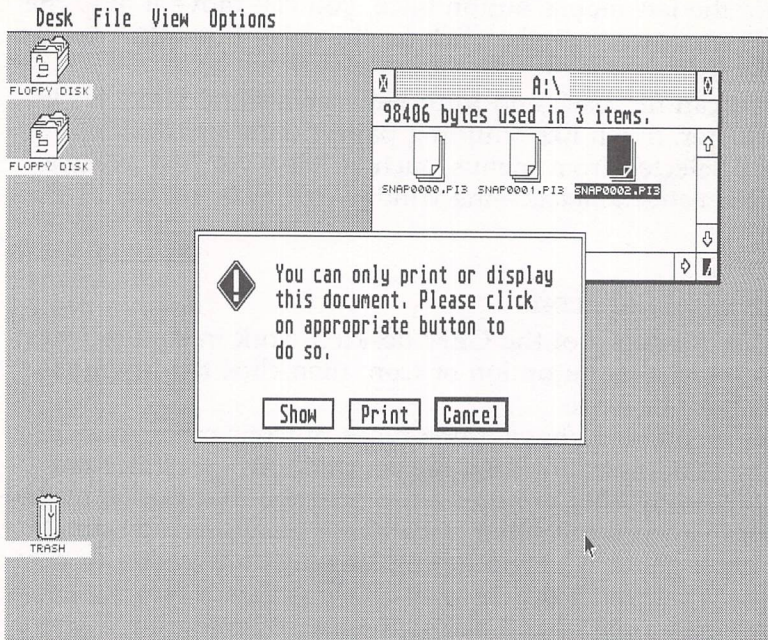
For instance, to run a program, you point to its icon or filename in the disk directory window and double-click. The desktop disappears and the program runs. When you exit the program, the desktop reappears.

The second type of icon which can appear in a disk directory window represents a *nonexecutable file*. The icon resembles a sheaf of papers with one corner folded back. This is a file that contains some kind of information, but it cannot be run as a program. For example, it may be a data file created with a spreadsheet program. To use the file, you must first load and run the spreadsheet, and then load the data file from within the program. (There is one exception to this rule, which we'll cover later.)

If you double-click on a nonexecutable file, something known as an *alert box* appears. Alert boxes usually pop open when you've attempted an action that is either not permitted or is irrevocable. In this case, you've attempted to run a nonexecutable file, so the alert box informs you and presents three options: Show, Print, and Cancel (Figure 4).



**Figure 4. Alert Box**



If you select the Show option, the computer will try to display the contents of the nonexecutable file on the screen. If the file is a text file created with a word processor, for instance, you may be able to browse through the document in this manner without loading and running the word processor. The computer displays a screenful of text, and you can press the Return key to scroll through the file a line at a time. Or press the space bar to scroll through a whole screen at a time. Pressing the Control and C keys together aborts the Show option and returns you to the GEM desktop.

If you select the Print option, the computer tries to dump the file to your printer.

Often, however, Show or Print will result in nothing but a screenful or pageful of seemingly random characters. Data files which are not pure text files are usually not meaningful when viewed outside the context of the application programs which created them. So if you don't want to Show or Print the nonexecutable file, you can select the Cancel option to make the alert box disappear and return you to the GEM desktop.



To select any of these options, point to the corresponding small rectangle within the alert box and click the left mouse button. These small rectangles—which in this case are labeled Show, Print, and Cancel—are known as *buttons*. (Don't confuse onscreen buttons with the physical buttons on the mouse controller.) As you'll soon discover, buttons within alert boxes are used for many purposes on the ST.

### Subdirectories

The third type of icon that can appear within a disk directory window is called a *folder*. It looks just like a manila file folder, or it's denoted by a diamond character if your directories are displayed in text format. Either way, a folder is a *subdirectory* on the disk. (The terms *folder* and *subdirectory* are interchangeable on the ST.)

If you've ever used MS-DOS on an IBM PC or compatible, you're probably familiar with subdirectories. In effect, a subdirectory is a disk directory within a disk directory. Today's large-capacity floppy disks—not to mention hard disks—would be unmanageable if it weren't for subdirectories.

Imagine, for example, that a secretary using a word processor is storing business letters on a floppy. Each letter might be only about 1K long. That means more than 300 files would fit on a single-sided ST disk, and more than 700 on a double-sided disk. Sifting through that many entries in a disk directory to locate a certain file would be laborious, to say the least.

Subdirectories help solve that problem by partitioning the disk into multiple directories. For example, the secretary could create a subdirectory named PURCHASE.LET for letters related to purchasing. Another subdirectory could be called CONTRACT.LET for letters related to contracts, and so on. Each directory is almost like a separate disk. You can load and save files within directories, delete files in directories, and copy files from one directory to another. You can have two files with the same filename stored in two different directories—something which normally isn't allowed on a disk.

It's even possible to create subdirectories within subdirectories. For instance, the CONTRACT.LET directory could contain a subdirectory of its own called SMITH.LET. This might contain copies of all letters related to contracts written to Mr. Smith. And SMITH.LET could contain yet another subdirectory called JULY.LET, holding copies of letters written to Mr.

Smith in July. The intelligent use of subdirectories helps keep your disks organized and your GEM desktop uncluttered.

### Opening and Creating Folders

Viewing the contents of a subdirectory is much like opening a disk directory window. You point to the folder icon or subdirectory name and double-click the left mouse button. In a few seconds, the folder opens and replaces the disk window with another window that shows what's in the folder.

To create a folder on a disk, first click on the disk's file drawer icon or directory window to indicate which disk should receive the new folder. Then drop down the File menu and select the New Folder option. An elaborate type of alert box known as a *dialog box* opens up. The dialog box contains a dotted line and a cursor so you can type in the name of the new folder. (Folder names must conform to the rules for file-names: up to eight characters plus an optional three-character extender.) Press Return or click on the OK button to create the folder. Click on the Cancel button if you want to abort the operation.

Incidentally, when working with folders, you'll probably encounter references to such terms as *root directory* and *directory pathname*. The root directory is simply the main directory of a disk, the one which appears when you first open the file drawer. Subdirectories are considered to be branches from the root directory.

A directory pathname is simply a way of specifying the disk and subdirectory (or subdirectories) in which a certain file can be found. For instance, if a file called TEST.DOC is on the root directory of disk drive A, the full pathname is merely A:TEST.DOC. If a file is in a subdirectory, the backslash character (\)—found on the keyboard next to the Return key—indicates a subdirectory name. So the pathname A: \CONTRACT.LET \TEST.DOC means that the file TEST.DOC is within the CONTRACT.LET folder on disk drive A. It's important to know how to specify a directory pathname when loading and saving files with application programs.

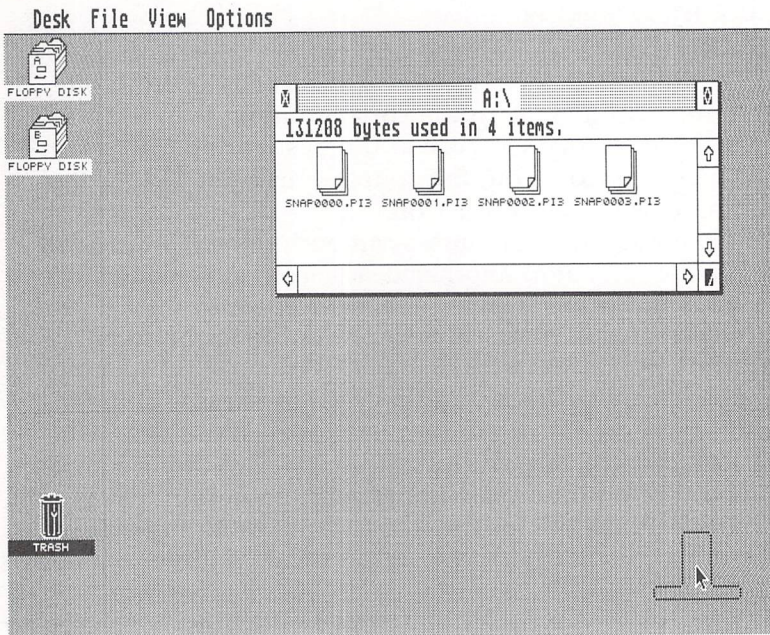
### More Mouse Dexterity

In addition to clicking and double-clicking, another mouse maneuver to master is known as *dragging*. This technique makes it possible to move icons on the GEM desktop.



One application for dragging is to rearrange the GEM desktop to suit your taste. For instance, normally the desktop appears with the file drawers for disk drives A and B in the upper left corner of the screen, and the trash can icon in the lower left corner. Suppose you want to move the trash can to the lower right corner. Simply point the cursor at the icon; then click *and hold* the left mouse button. A dotted outline of the icon should appear. Now, while still holding down the mouse button, you can move the dotted outline anywhere on the desktop. If you move it to the lower right corner of the screen and release the button, the trash can icon reappears at that position (Figure 5).

**Figure 5. Rearranging the Desktop**



You can move the file drawer icons in a similar fashion. Note, however, that GEM won't let you move certain icons to certain places. If you try to move a file drawer onto the trash can, an alert box tells you that you can't delete an entire disk in that manner. Nor will GEM let you move the trash can into a disk directory window.



Other dragging operations imply that you want to perform certain file-management functions. They take the place of commands typed at prompts with operating systems like MS-DOS. To delete a file or folder from a disk, for example, drag the corresponding icon to the trash can.

Note that, unlike the Macintosh and Amiga trash cans, the ST's trash can is more like an incinerator; deleted files cannot be recovered. Therefore, GEM pops open an alert box which gives you a chance to abort this potentially destructive operation. It's also important to remember that trashing a folder deletes *all the files* within that folder. The alert box will always warn you of the number of items you're about to delete.

### Copying by Dragging

You can also copy files, folders, and entire disks by dragging icons. There are two ways to copy a file or folder from one disk to another: Either drag the corresponding icon from the source disk's directory window to the destination disk's window, or drag the icon from the source's disk window to the destination disk's file drawer icon.

You can copy the contents of an entire disk by dragging its file drawer icon atop another disk's icon. And you can copy a file into a folder by dragging the file icon onto the folder icon. (Note that this makes a *copy* of the file in the folder, leaving the original file intact.)

Windows are as easily manipulated as icons. The active window—the one on top of the pile if several are displayed—has various control bars and miniature buttons along its edges. Pointing to the button in the upper right corner and clicking the mouse button expands the active window to full-screen size. Clicking this button again restores the window to its original size.

The button in the upper left corner of a window is used to close the window. (Another way to close a window is to select Close Window from the File menu.) If you close a window that shows the contents of a folder, it is replaced by the next higher branch in the directory hierarchy. That is, if the folder you're closing is contained within another folder, then the "higher" folder's window will be displayed next. Eventually, you'll end up with the root directory window displayed on the screen.

Dragging the button in the lower right corner of a window lets you adjust the window's size, making it larger or smaller. Dragging the top bar lets you move a window anywhere on the screen. Clicking on the small arrows displayed along the bottom and right edges will scroll the material displayed in the window, assuming some of it is hidden due to the window's small size.

### Multiple Selection

We've covered all the main features of the GEM desktop, but several additional techniques make the system even easier to use.

One handy trick is *multiple selection*. This technique lets you select more than one icon at a time to perform a certain function in a single step. For instance, let's say you want to delete four files from a disk. You could drag them one at a time to the trash can. But with multiple selection, you can drag all four of them at once.

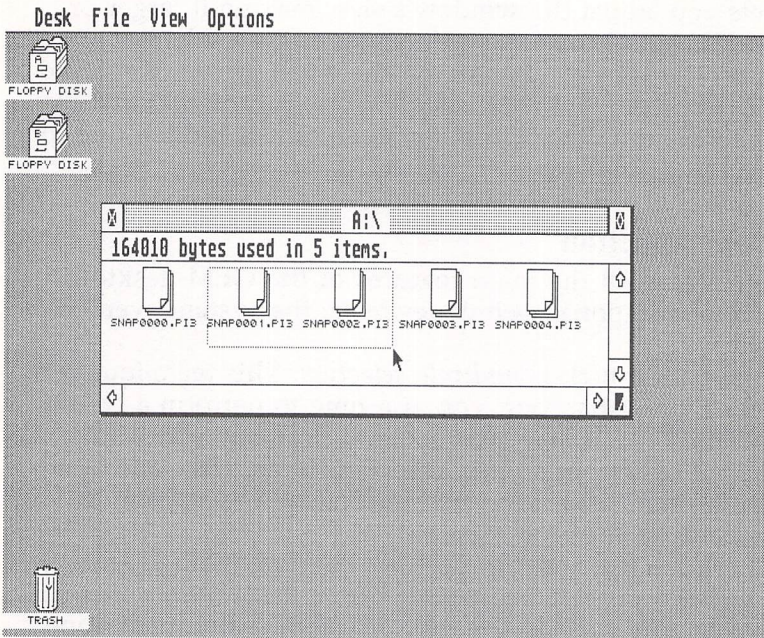
There are two ways to do this. The first method is sometimes called *roping*. It works only when the multiple files are grouped together inside the disk directory window. To rope the files, move the mouse pointer to the upper left corner of the file icon or filename which is at the *upper left position* of the files you want to rope. Then press and hold the left mouse button. A small dotted line should appear. Next, still holding down the mouse button, drag the pointer downward and to the right. The dotted-line "rope" should expand as you drag. Keep expanding the rope until it encloses all of the files you want to select, excluding any others. Then release the mouse button (Figure 6).

When you release the button, the file icons or filenames enclosed by the rope should change color to indicate they've been selected. Now you can drag them all at once to the trash can, just as you would drag any icon: Press and hold the left mouse button and drag the files across the desktop. Release the button to drop the files in the trash (Figure 7).

Obviously, roping works for multiple selection only when the files are grouped together inside the window. If the files are scattered among other files in the window, there's no way to rope them together. That's when you need to use the second method for multiple selection: Simply hold down either



**Figure 6. Roping**



Shift key, point to a file you want to select, and click the left mouse button. The file should change color to indicate it's been selected. You can repeat this process as many times as necessary, as long as you keep holding down the Shift key. When you've selected all of the desired files, drag them in the usual fashion to the trash can.

Either method of multiple selection is good for other things besides deleting files. You can also use the technique to copy several files from one disk to another or from one directory to another.

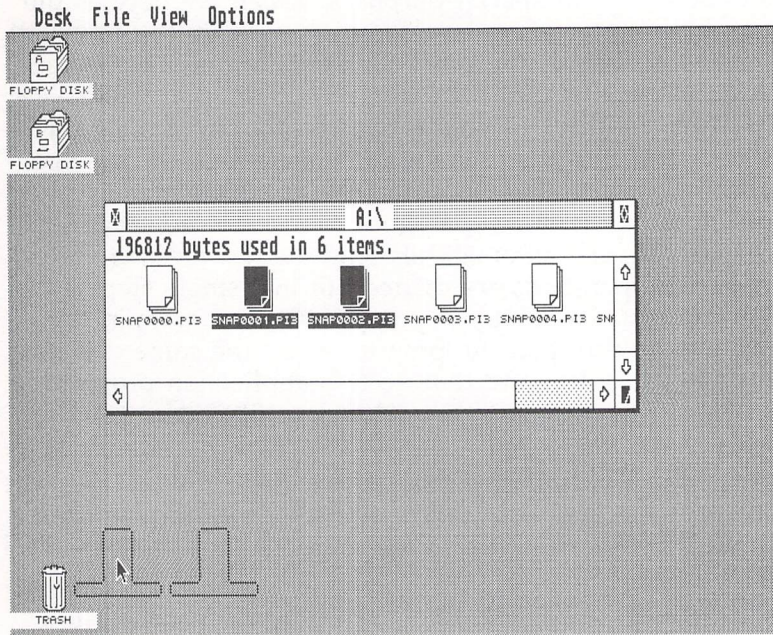
### Installing an Application

Another GEM technique lets you make an exception to the rule that nonexecutable data files can't be loaded and run like program files. It's called *installing an application*.

Let's say you've got a word processing program on your disk and a number of text files that were created with the word processor. Normally, to edit one of the text files, you'd have to load and run the word processing program by double-



**Figure 7. Dropping Files into the Trash**



clicking on its file icon or filename and then load the text file from within the word processor. But there's a shortcut.

First, click once on the icon or filename of the application program (in this case, the word processor). It should change color to indicate it's been selected. Then drop down the Options menu and select Install Application. A dialog box appears. Among other things, you'll see a prompt that reads *Document Type:*, followed by a dotted line and cursor. Here's where you type in a three-character filename extender that you'll use to identify all of the data files associated with this particular application program (Figure 8).

For example, you might decide to identify all of your text files with the filename extension .TXT—SMITH.TXT, REPORT.TXT, LETTER.TXT, MEMO.TXT, and so on. In that case, you'd enter .TXT at the *Document Type:* prompt.

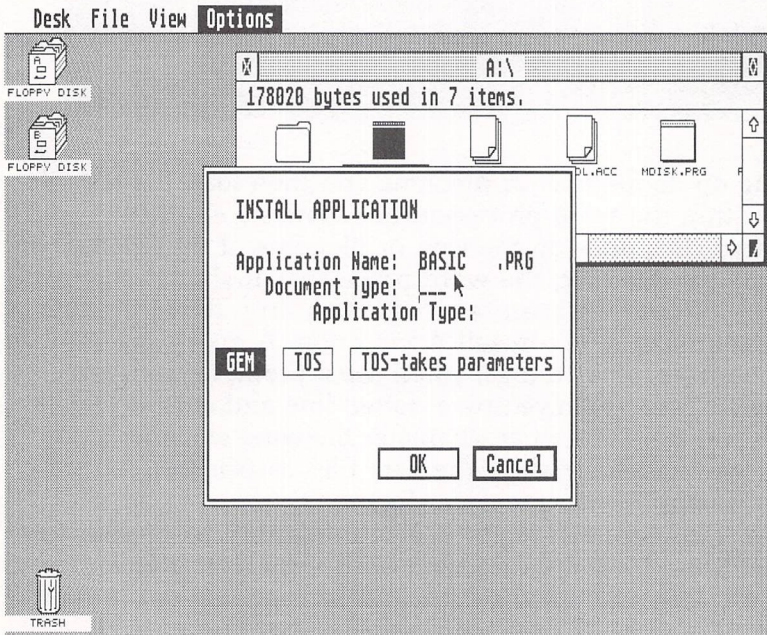
Below the prompt, you'll see three buttons labeled GEM, TOS, and TOS-takes parameters. If the application program you're installing supports GEM features—in other words, if it has drop-down menus and so forth—click on the GEM but-

ton. Otherwise, click on the TOS button. (The TOS-takes parameters button is for special purposes, which are beyond the scope of this chapter.) Finally, click on the OK button to install the application. The dialog box goes away, and you're back on the GEM desktop.

To enjoy the fruits of your labor, try double-clicking on one of your text files that has the filename extension you specified at the *Document Type*: prompt. (Make sure the word processor is on the same disk.) If all goes well, the word processor should load, run, and automatically load the text file you clicked on. Thus, two operations are carried out in a single step.

You can install virtually any application program in this manner. Just be sure that the program is on the same disk as the data file you select and that no two application programs are installed with the same three-character filename extender.

**Figure 8. Identifying Data Files**





To keep from having to install the application each time you switch on your ST or press the Reset button, insert your boot disk in drive A, drop down the Options menu, and select Save Desktop. This creates a short file on your boot disk that GEM checks for each time it boots up. The file contains information that lets GEM restore the conditions which existed on the desktop when you created the file. Before selecting Save Desktop, you might want to arrange the disk directory windows and icons as you'd prefer to see them appear when the computer first comes to life.

### Three Screen Modes

One unusual feature of the ST is its intelligent monitor interface. When you boot up the computer, the operating system checks whether a monochrome or color monitor is attached and adjusts itself for one of three possible screen modes.

With the monochrome monitor, the operating system automatically configures the GEM desktop for high resolution—640 × 400 pixels, black-and-white. The display is extremely sharp and stable because of the monitor's 70-hertz refresh rate, which means it redraws the screen image 70 times per second rather than 60 times as on standard monitors and TVs. (This is possible because the monitor uses its own 70-hertz oscillator instead of synchronizing with the 60-hertz power line.) Furthermore, the display is paper white, not blue white, easier on the eyes. When the monochrome monitor is hooked up, the operating system won't let you enter the medium- or low-resolution mode, which has color.

If the ST is booted up when plugged into its RGB color monitor, it defaults to the medium- or low-resolution color mode. Medium resolution has 640 × 200 pixels with four simultaneous screen colors, and low resolution has 320 × 200 pixels with 16 simultaneous colors. Because the medium-resolution screen has the same horizontal resolution as the monochrome mode, but only half the vertical resolution, the aspect ratio is slightly distorted. Icons appear tall and skinny, and characters are narrower. In low resolution, only 40 columns of text can be displayed instead of 80 columns. (See Figures 9, 10, and 11.)

## CHAPTER ONE

Figure 9. Low Resolution

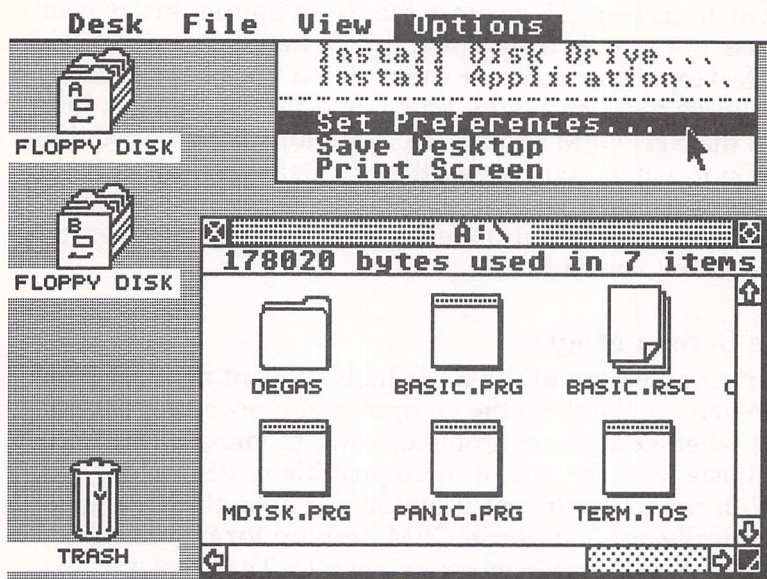
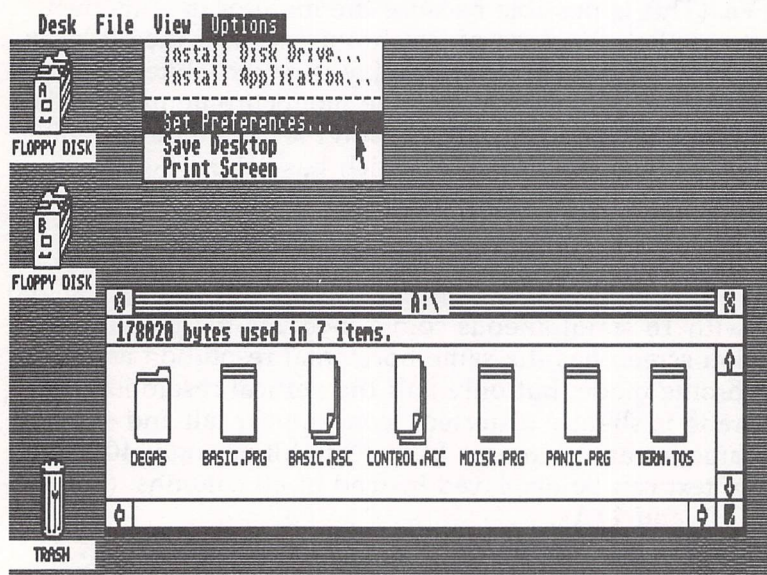
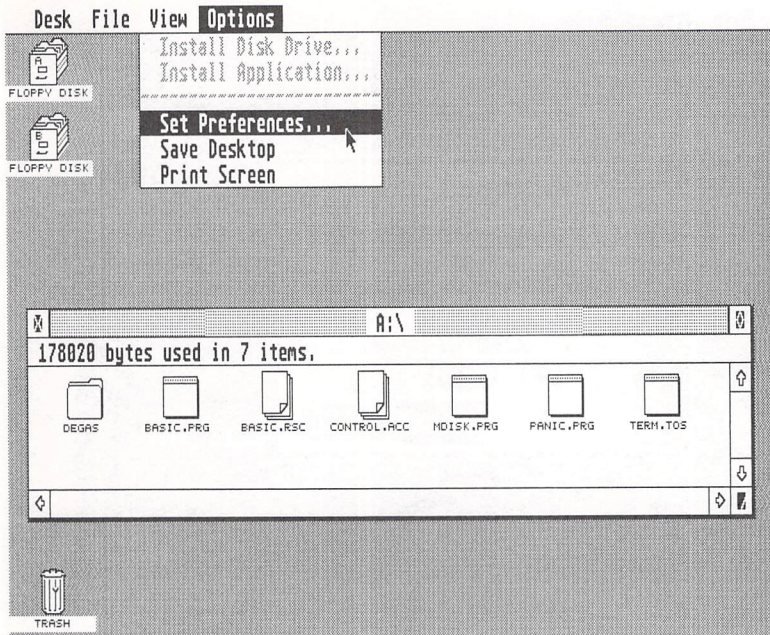


Figure 10. Medium Resolution





**Figure 11. High Resolution**

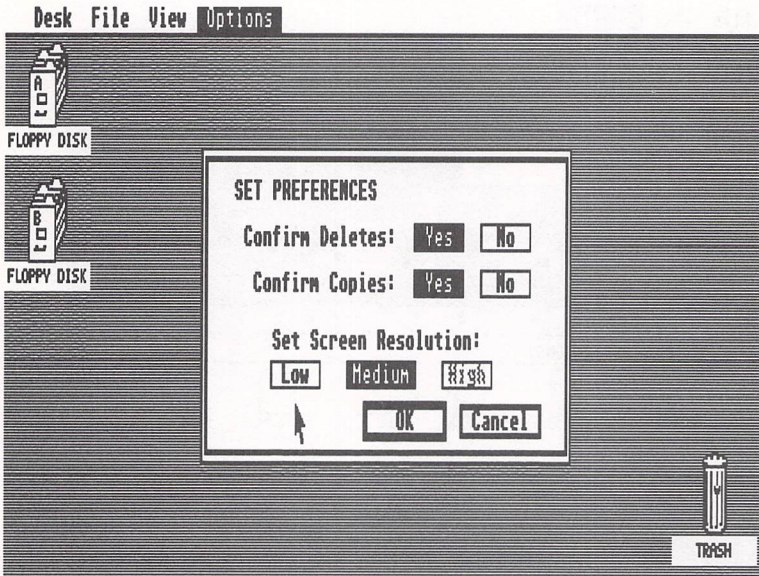


The low-resolution mode is ideal for graphics programs that need the maximum number of colors and for STs that are hooked up to a TV set instead of an RGB monitor. (Only the 520STs made after the spring of 1986 have RF modulators for TV hookup.)

To change from one screen mode to another, drop down the Options menu and select Set Preferences. A dialog box appears, and a prompt that reads *Set Screen Resolution:* is accompanied by three buttons labeled Low, Medium, and High. If the ST is hooked up to a color monitor, the High button is dimmed to indicate it's disabled. If the ST is hooked up to a monochrome monitor, the Low and Medium buttons are dimmed (Figure 12). Click on the button for the screen mode you want, and then click the OK button.

Incidentally, this dialog box also lets you choose whether GEM will warn you with an alert box whenever you try to copy or delete a file. Just click on the Yes or No button next to the prompts *Confirm Deletes:* and *Confirm Copies:*.

Figure 12. Setting Screen Resolution



If you want your ST to “wake up” in a certain color mode when you switch it on, adjust the screen for that mode with Set Preferences and then select Save Desktop.

### The Control Panel

GEM lets you make other adjustments to your desktop as well, thanks to a tool called the Control Panel.

The Control Panel can be found on one of the disks that came with your ST, usually under the filename CONTROL.ACC. It's a *desk accessory* that installs itself in GEM's Desk menu whenever you boot up with that disk.

What's a desk accessory? It's a program that's specially designed to load itself into memory when you first switch on the ST. But after loading, it doesn't run—it just waits invisibly in the background until it's summoned. To call a desk accessory, you select it from the Desk menu. The Desk menu is always available from the GEM desktop and from within any application program that supports GEM. (In many application programs, the Desk menu is labeled with the Atari logo symbol instead of the word *Desk*.) Your ST comes with a few desk



accessories, including the Control Panel and a VT-52 terminal emulator. Many other desk accessories are available from software companies and other sources. Some of them are quite elaborate.

A desk accessory is a limited form of multitasking. When you call up an accessory, it appears on the screen and runs without interfering with any other application program you happen to be using. When you close the desk accessory, you can resume using the application program and pick up where you left off.

The Control Panel is a special desk accessory that lets you adjust certain characteristics of the GEM desktop. To see how it works, switch on your ST with the boot disk that contains CONTROL.ACC. When you drop down the Control Panel, you should see two accessories installed: Control Panel and Install Printer. (The Control Panel is actually a double desk accessory that always includes Install Printer.) Select the Control Panel as you would any menu item.

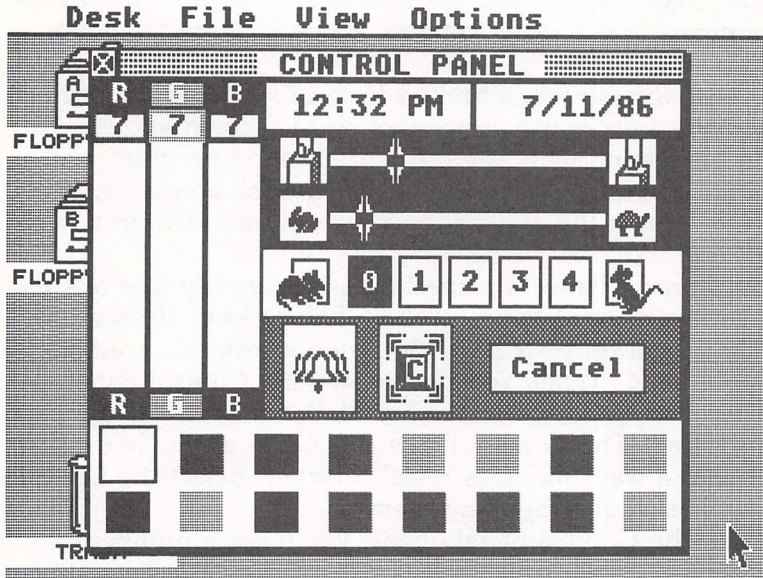
When the Control Panel opens, you'll see a number of buttons and slide controls (Figure 13). At the far left and bottom of the Panel are three controls for adjusting the screen colors. Of course, these have no effect in monochrome mode. If your ST is set up in a color mode, try changing the screen colors by manipulating these controls.

At the top of the Panel are dotted lines where you can enter the current time of day and date. To enter these numbers, click on the appropriate dotted line, type in the correct information, and press Return.

Below the time/date fields are slide controls which adjust how soon a key begins autorepeating when it's held down and how fast it autorepeats. To adjust these controls, point to the appropriate slider and hold down the left mouse button. Move the slider to the right or left, and release the button. The picture of the turtle indicates the direction to move the slider for a slower rate, and the picture of the rabbit indicates the direction for a faster rate.

Below the slide controls are a series of buttons numbered 0-4. These let you adjust how rapidly the left mouse button must be pressed to register a double-click. The slowest double-click speed is 0 (next to the picture of the lazy mouse); the fastest double-click speed is 4 (next to the active mouse). You might want to pick a slower speed until you get used to your ST.

Figure 13. The Control Panel



Two more buttons below this let you determine whether keypresses will return audible feedback through the monitor speaker and whether error messages will be accompanied by a bell tone. Click these buttons on or off as desired.

Finally, when you've finished making adjustments, close the Control Panel by clicking on the small button in its upper left corner. If you messed something up and want to abort the changes you made, click on the Cancel button near the lower right corner instead.

Adjustments made with the Control Panel remain effective until the computer is rebooted. If you want certain adjustments to be made automatically whenever the ST is switched on, drop down the Options menu and select Save Desktop after using the Control Panel.

### Secrets of GEM

This chapter isn't a complete treatment of the GEM desktop, but it does cover the majority of important features. Additional information can be found in the manual which came with your ST.



Some features of GEM aren't documented in the manual, however. For example, many people wonder *What is that useless right mouse button for?* True, some application programs use the right button, but GEM doesn't seem to. The secret is that the right button lets you perform operations in an inactive window without making it active.

For instance, try opening two disk directory windows on the desktop. Point to a file within the inactive window while holding down the right mouse button. Then click the left mouse button. The file will change color to indicate it's been selected, even though the window is still inactive. You can now drag the file to the trash can to delete it, drag it to the active window to copy it to the other disk, or even run it if it's a program file. It's no big deal, but it is a little-known secret.

As you continue to explore the ST and its GEM desktop, no doubt you'll discover other tricks and shortcuts as well.

# ST System Software, Inside Out

Bill Wilkinson

---

*The ST's operating system and graphics-processing environment are loaded with acronyms—TOS, BIOS, XBIOS, GEMDOS, VDI, and AES. Here's a quick look at what they mean.*

Okay, you've got your shiny new ST computer plugged in and running. You can use the mouse to select programs, copy files, and format disks. It's fun, and it certainly is easier to learn than figuring out what

**COPY B: \SYSTEM \MSG.S.TXT/A=A:  
SPCL\*.MS?**

is supposed to mean. That's a real and possible IBM PC command. But how did the ST system get built?

Collectively, the software built into the Atari is called TOS (Tramiel Operating System). When the 520ST was first shipped, TOS was delivered on a disk. If you're still using the disk-based TOS, stop now. Go out and buy the ROM (Read Only Memory) version of TOS. It should cost no more than \$25 or so. Installation is not very difficult, but if you have as many left thumbs as I do, you might be well advised to find a dealer or service center to install the chips for you. That will cost maybe another \$20 to \$30.

TOS in ROM is actually composed of six separate pieces. Usually, we lump these pieces into two groups of three each: the graphics-processing section and the underlying operating system. As we shall see, that operating system—a derivative of CP/M-68K—is very similar to MS-DOS and PC-DOS, which are both derivatives of CP/M.

## **BIOS, XBIOS, and GEMDOS**

In one sense, we can say that the lowest level of the ST's operating system is the BIOS (Basic Input/Output System), a holdover from the earliest days of CP/M. At this level, we find routines for such basic tasks as sending a single character



to a device, reading a disk sector (by sector number—a *very* dangerous practice), and so on. In CP/M, there was only one legitimate reason to call the BIOS directly: speed. With TOS, though, only the BIOS provides some of the facilities which even a moderately sophisticated program will need (admittedly, often because of bugs in the upper levels of the operating system).

On the ST, a BIOS call is implemented as a TRAP instruction in 68000 machine language. All the necessary parameters, including the BIOS call number, are passed onto the stack. If you aren't quite sure what we're talking about, don't worry. Virtually every programming language for the ST has some way to use these routines which mask the mechanics of TOS calls. It's a good thing, too, since some of those mechanics can get pretty hairy.

The next higher component of TOS is the XBIOS (extended BIOS). XBIOS supplies the Atari-unique routines needed to do such things as access the sound registers, screen hardware, and so on.

The third component of the operating system is called GEMDOS (Graphics Environment Manager/Disk Operating System). Actually, this is a misnomer. The GEMDOS routines have nothing whatsoever to do with graphics. GEMDOS is essentially an MS-DOS or PC-DOS emulator. Want to open a file? Read a block of bytes? Get a character from the keyboard? Given the differences between the 68000 of the ST and the 8088 of the IBM PC, the similarities between GEMDOS calls and MS-DOS calls are almost scary.

### GEM, VDI, and AES

Okay, enough about the underlying operating system. Let's take a look at the graphics systems which comprise GEM. The most familiar part is the *GEM desktop* which appears when you turn on your ST. But the desktop is not really a special program at all; it simply calls the lower level routines. Again, there are three levels of graphics routines.

The lowest-level graphics, not officially part of GEM, but merely one means of implementing it, are those called the *Line-A Routines*. This sounds cryptic, but it simply refers to the fact that certain machine instructions of the 68000, including those of the form \$Axxx hex (hence, line-A), are reserved and

cause a special hardware trap into the OS. As you might expect, routines implemented in this fashion are of the most fundamental type: they draw a line, plot a point, and so forth. Most are very fast.

The next level up in graphics is the *VDI* (Virtual Device Interface). In theory, VDI is capable of supporting several types of graphics devices in a uniform fashion. For example, you might use the same set of calls to draw a curve on a plotter or on the screen. Unfortunately, no such drivers are yet available (or, as far as I can tell, are even in the works) for the ST. Still, the possibility exists.

VDI does all the actual graphics work on the ST. It draws simple rectangles, bordered ovals, and text in various styles, sizes, and colors. Someone who learns nothing on the ST except how to call VDI could still do remarkable graphics work.

Finally, at the highest level, is *AES* (Application Environment Services). AES is what GEM uses to present you with that nice, pretty desktop, complete with menus, dialog boxes, alert boxes, windows, and icons. Perhaps more important to programmers, though, is the fact that AES allows you to use all the features of GEM in a relatively consistent, properly desktop-compatible manner. It is through this mechanism that even a lowly spreadsheet program can have drop-down menus, mouse-controlled windows, and all the rest of those impressive features.



# Odd Facets of GEM

Bill Wilkinson

---

*This article explores some of the GEM desktop oddities—things you can do to make your system more useful, things you can do to make your system crash.*

I'm a natural-born pessimist, so let's start with a GEM desktop crash. The bug we're about to demonstrate infests the ROM (Read Only Memory) version of the TOS operating system. Even with this problem, if you don't already have the TOS ROMs, get them today. The difference in overall system performance and capability is only a little short of great.

To see this bug, simply boot your system and bring up the Control Panel. Select either the date or time field. Then type an underline character (Shift-hyphen) and watch your system bomb. The only recovery is to press the Reset button or turn off the power.

**Problem:** The Control Panel is a form of dialog box, and it uses what are known as *editable text fields* to display and let you modify the date and time. An editable text field is designed to restrict the user to typing certain characters. For example, the date and time fields of the Control Panel are editable fields which allow only numbers to be typed. Unfortunately, somehow a bug crept into the ROM-based TOS. Anytime you edit a strictly numeric field, typing the underline causes something nasty to occur. Editable fields for filenames have a similar, though usually nonfatal, problem.

**Solution:** In a GEM application program that needs to accept numeric-only input from the user, there are two choices: Use an editable field that allows *any* character, and validate the user's input after the dialog box has returned, or retrieve keystrokes one at a time (checking them on the fly), and print only the valid ones on the screen. The first solution is ugly because the user doesn't get immediate response to incorrect input. The second solution is a lot of work. Take your pick.

### Modifying DESKTOP.INF

You may already know how to customize the GEM desktop so that your preferences appear automatically when you boot up the ST. When you select Save Desktop under the Options menu, GEM saves a file to the currently active drive, called DESKTOP.INF, which stores these preferences. You can rearrange the icons on the screen, change screen colors, resize the windows, and so on, and GEM remembers it all for you.

DESKTOP.INF is an ordinary ASCII file, so it can be modified with most text editors and word processors. This lets you personalize GEM even more (see “ST Hints and Tips” elsewhere in this book).

The first thing we’ll do is the easiest. Using a text editor or word processor that handles ASCII files, load and examine DESKTOP.INF. You should see one or two lines which contain the words *FLOPPY DISK* (among other things). These are the labels which appear beneath the disk icons. I usually rename the labels *-Top-Disk-* and *Bottom-Disk* (the dashes indicate where I typed a space since typesetting makes it hard to show spaces).

Save the modified file back on disk in ASCII format. The next time you boot from that disk, the names should appear as you have modified them. Just for fun, sometimes I change the name of the trash can to *Junk!* or *Garbage* or something equally silly.

### Rearranging Files

There are even more interesting things you can do with DESKTOP.INF. If, like me, you have a disk or subdirectory in which you do most of your work, you’ll soon find that you can’t see all of the filenames or icons on the screen at once. Although it’s a minor nuisance, it always seems that the files (or, more likely, programs) that I want the most are always off the screen. How can they be forced back on the screen (preferably in the upper left position)?

One solution, since the default display mode under the Show menu is *Sort by Name*, is to name your favorite files AARDVARK.PRG or AAABASIC.PRG. But that’s messy. A better method might be to choose *Sort by Date* if you could change the file’s creation date. But I think Mark Rose, of Optimized Systems Software, has hit upon the best scheme.



First, he chooses *Sort by Type*. Second, he renames his most-used programs so that they have no type (filename extension) at all. Third, he loads DESKTOP.INF and adds a line or so. To figure out exactly what to add, look for a line in DESKTOP.INF similar to this:

```
#G 03 FF *.PRG@ @
```

This line tells the desktop that all files which match the \*.PRG specifier are GEM (G) program files. Now, let's say the program you want to appear at the top left of the screen was called PASCAL.PRG and has been renamed simply to PASCAL. You would add this line to the end of the DESKTOP.INF file:

```
#G 03 FF PASCAL.@ @
```

This tells GEM that PASCAL is actually a GEM-based program. Neat, huh? What's more, you can do this for several files. However, I *do not* recommend using the \*. wildcard in such a line. General untyped files end up looking like programs—a dangerous deception.

# GEM Quirks

Bill Wilkinson

---

*These suggestions offer application programmers help in dealing with system software problems, speeding up disk drive performance, and avoiding GEM crashes.*

The Atari ST's hardware is excellent, but all too often, problems with its system software obscure this excellence. Admittedly, most users will never actually see these problems, since software developers work hard to circumvent them. Luckily, application programmers can make a real contribution to the users' perceptions of a machine.

Consider the ST's floppy disk drives. In theory, they are among the fastest available for any microcomputer. And, indeed, when you load a program, the speed is impressive. However, when a program starts performing file input/output using ordinary record sizes, there is so much operating system overhead to overcome that the ST's performance is only fair. Creating a new file with 512-byte records is only a little more than twice as fast on an ST as it is on an Atari 400/800, XL, or XE.

Possible solution: The application program can read and write very large blocks to the disk (for example, 4K or bigger), performing the file buffering itself. Suddenly the performance is quite good again. This requires a little more work on the part of the application programmer, but the net effect is pleasing for the user.

Similarly, using a hard disk on the ST is an experience not to be forgotten. For example, compiling an average-length program with *Personal Pascal* generally takes one to two minutes using floppies. When you use a hard disk, those times improve to 10 or 15 seconds. That's because the hard disk port on the ST is capable of transferring more than one megabyte per second.

But something happens as the hard disk starts filling up. Access times can double before the disk is even half full. Again, there's a solution: Partition the 20-megabyte disk into four smaller, 5-megabyte "logical" drives. And, since the ST



uses subdirectories so successfully, this is usually a practical solution.

### Gullible GEM

Perhaps the biggest problem with GEM (the Graphics Environment Manager) is that it is too gullible; tell it a lie and it believes you. Consider what happens on an Atari 400/800, XL, or XE when an Atari BASIC programmer uses a PRINT statement to display a message which is wider than the screen: The text wraps around to the next line.

When programming with GEM, the easiest way to display something on the screen is via an *alert box*. This is the small window which pops up to report errors and so forth. To display an alert box, a programmer simply defines a string of the proper form and makes an easy call to a GEM routine. But if the programmer errs when defining that string (for example, by entering too many characters or leaving out some special characters)—*crash!* Time to hit the old Reset button.

Now, granted, the proper form of that string is easy to validate before calling GEM, so a well-written application program will never reveal this particular problem to its user. However, this is symptomatic of much of GEM. Application programmers must do a lot of work to insure that GEM is given only legal values to work with. GEM does not seem to follow the *GIGO* rule (Garbage In, Garbage Out). With GEM, it is more like *GIC* (Garbage In, Crash!). So be careful if you're writing programs on the ST. Avoid crashes by double-checking all data before calling GEM routines.

### The Software Explosion

To a beginner, the ST with its GEM operating system looks complex. And, truly, there is a *lot* to learn before you can write programs which show off all the capabilities of the ST. But, despite my earlier comments, experienced programmers find that GEM does so much of the work for them that they can develop fairly complex programs relatively quickly. Too, the capabilities and accessibility of higher level languages for the ST (such as C, Pascal, and Modula-2) have made programmers more productive. As a result, there is arguably more software available for the ST, at this point in its life, than for any previous computer at a comparable point in its life.

For instance, one year after the Macintosh was introduced, it had far fewer programs available than the ST had a year after its introduction. Not only that, but the ST programs have tended to be considerably less expensive than their Macintosh counterparts.

One of the reasons so much software is appearing is that the cost of developing ST programs is relatively low. A part-time ST programmer can have a full-blown ST development system for not much over \$2,000 (including hard disk, printer, color and monochrome monitors, development software, and so forth). In the early days of the Mac, \$10,000 was more the order of the day, so development tended to be restricted to established software companies.

The flip side of this coin is that the quantity of high-quality software for the ST is certainly *not* greater than what was available for the Macintosh. Since most early Mac developers were major software companies, their quality standards were generally higher than those of part-time hackers.

Bottom line: Try to see a demo of any ST software you are planning to purchase. There are a lot of excellent ST programs, but there are also some turkeys.



# CHAPTER TWO

---

# Games





# Switchbox

Todd Heimarck

Version by Kevin Mykytyn

---

*At first, you may think this challenging game of strategy looks easy and simple, but it takes time to master and permits many variations.*

Playing “Switchbox” is like putting dominos in place for a chain reaction—either you’re setting them in position or you’re knocking them over. Winning requires skill and a sense of when to go for points and when to lie back and wait for a better board. The goal is simple: You try to score more points than your opponent by dropping balls into a boxful of two-way switches. Each switch has a trigger and a platform. If the ball lands on an empty platform, it stops dead. But if it hits a trigger, it reverses the switch and continues. In many cases dropping a single ball creates a cascading effect—one ball sets another in motion, which sets others in motion, and so on, all the way down.

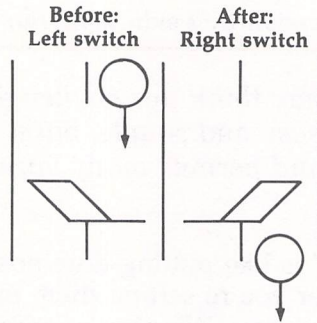
Type in the program, and save a copy before you run it. Before typing it in, you must turn off the buffered graphics by clicking on Buf Graphics in the Run menu. You can be in either medium- or low-resolution mode when you type in the program, but before running it, you must be in low-resolution mode. Do this by selecting Set Preferences from the Options menu on the desktop and clicking on Low.

## A Box of Switches

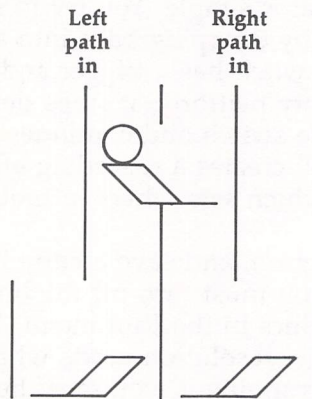
Switchbox is a tale of twos: Each switch has two parts, two positions, two states, two paths in, and two paths out. The two parts are the platform and the trigger. A switch can lean to the left (platform left, trigger right) or to the right (platform right, trigger left). See Figure 1.

The trigger is weak, and it always allows balls to pass. But the platform is strong enough to hold a single ball. So the platform either holds a ball—it’s full—or it does not and is empty. When a ball sits on a platform, the switch is said to be loaded, or full.

**Figure 1. Trigger States**



**Figure 2. Loaded Trigger**



In Figure 2, a full switch is over two empty switches. The platform holds a ball and leans to the left. The trigger extends to the right. Notice that the switch on top has two pathways leading in, the left path and the right. The right path leading out is the left path into one of the switches below. The left path of the top switch leads into the right path of the other, the switch below and to the left. If you drop a ball down the righthand path, it will hit the trigger and flip that switch to the right. Then it will continue down, hit the lefthand trigger below, and flip that switch as well.

In the meantime, the ball on the platform is set in motion (when the switch is flipped), and it hits the trigger. The top switch is reset to point to the left. The second ball then drops a level to the platform below, where it stops.

The playing field is composed of five levels, with four switches in the first level and eight in the bottom level. At the



beginning of the game, there are no balls on the field—all platforms are empty—and the position of each switch is chosen randomly.

### **Moving Down the Path**

Players alternate dropping balls into one of eight entry points. These balls (and others) may or may not make it all the way through the switchbox, to one of the 16 exit paths. Balls fall straight down (with one exception), so a ball's movement is always predictable. When it hits an empty switch, one of two things can happen. If it lands on the empty platform, it stops dead in its tracks. But if it lands on a trigger, it falls through to the next level below.

Moving balls always make it through loaded switches. Triggers allow balls to continue and move the switch to the other position. If it's loaded, the dead ball on the platform is put into motion, and it hits the trigger that just moved over. This makes the switch go back to its original position, but with an empty platform. So when a ball hits the trigger of a loaded switch, its motion continues unabated. The switch moves, the ball on the platform begins to fall, and it hits the newly placed trigger. The newly emptied switch moves back again, and the two balls drop to the next level.

There's one more possibility: a ball can drop onto a platform that already holds a ball. Since a platform can't hold more than one ball, when this happens, one of the balls slides over to the trigger. So the ball doesn't move straight down, but slides over to the next pathway. This is the exception to the rule that balls drop in a straight line. Of course, when the ball hits the trigger, the switch changes position, causing the other ball to drop and hit the trigger.

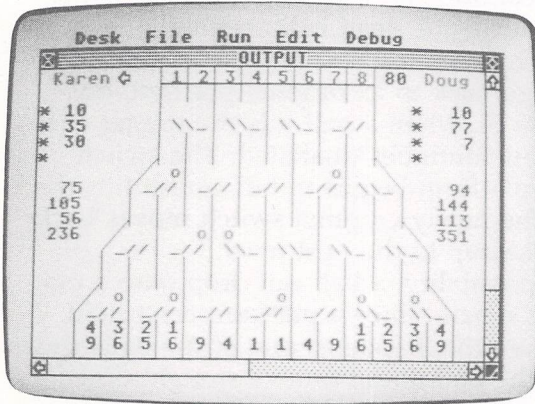
### **Chain Reaction**

At the start of the game, all platforms are empty, so four of the eight entry paths are blocked. Remember that your turn ends when a ball hits an empty platform and stops. As the switches fill up, the chances increase that a ball will descend through several levels. Your goal is to score points by getting balls to pass all the way through the maze of the switchbox. The best way to collect a lot of points is to cause a chain reaction.

## CHAPTER TWO

A ball that hits a loaded switch from either side continues on its way. And the previously inert ball on the platform starts moving. One enters, two exit. If both of those balls encounter full platforms, four drop from the switches. The pathways are staggered, so the effects can spread outward, with more and more balls cascading toward the bottom.

Rather than taking an easy point or two, it's often worth your while to build up layers of loaded switches. Watch out for leaving yourself vulnerable, though. Because players take turns, you'll want to leave positions where your opponent's move gives you a chance to create a chain reaction. Defensive play is the best strategy. Look ahead a move or two, and watch for an opening that will allow you to score several points at once.



*"Switchbox" is a challenging strategy game.*

### Four Quarters

A game of Switchbox always lasts four rounds. In the first (equality), each exit counts for 2 points. Your goal is to score 10 points. The second quarter has more points available as well as a higher goal. If you look at the exits, you'll see that the farther they are from the middle, the higher their point value. The numbers increase in a Fibonacci sequence: 1, 2, 3, 5, 8, and so on (each number is the sum of the previous two numbers:  $1+2$  is 3,  $2+3$  is 5,  $3+5$  is 8,...). The target score in the second round is 40.

In the third round, the numbers are a bit lower. They increase arithmetically: 1, 2, 3, 4, and up (to 8 in the corners). A



goal of 20 points brings you to round 4, where you can score big. Here the numbers are squares: 1, 4, 9, 16, all the way to 64 at the edges. In rounds 2–4, it's sometimes prudent to leave a middle path open for your opponent to score a few points. Then you can gather a high score on the big numbers to the left and right.

Each round lasts until one player reaches the goal. At that point, the other player has one last turn before the round ends. It's still possible for the losing player to win the round on this last-chance play. Watch out for barely topping the goal and leaving a chain reaction open for your opponent.

An arrow points to the scoreboard of the player whose turn it is. On the other side of the screen, you'll see a number where the arrow should be. That's the goal for the current round.

### **Bonuses**

At the conclusion of each round, bonus points are awarded. Four numbers appear below the scorecards. The first number is simply the total so far. The second is the total plus a bonus of the goal for the round if the player's points are equal to or greater than the goal. For example, if the goal is 20 and you get 18, there's no bonus. If you score 22, the bonus is the goal for that round (20), so you'll have 42 points. The third number under the scoreboard is the difference between scores for the rounds. If you win by 2 points, you'll have 2 added to your score, and 2 will be subtracted from the other player's score. The final number is the grand total of the first three scores and bonuses. Rounds 1 and 3 are fairly low-scoring with low goals. You may want to seed the field with extra balls during these quarters, so you can collect more points in the second and fourth quarters.

### **Variations**

Although the goal of the game is to score the most points, there's no reason you couldn't agree to play for low score. In a "lowball" game, you would try to avoid scoring points. You wouldn't necessarily play backward; you would have to adjust the strategy of where to place the balls. Fill up the board as much as possible and leave your opponent in a situation where he or she is forced to score points.

The DATA statements at the beginning of the program determine the goal for each round and the point values for the exit paths. You can prolong the game by doubling the goals. This also dilutes the value of a big score at the beginning of a round, preventing one player from winning on the first or second turn. An interesting variation is to assign negative values to some slots. If some paths score negative points, you're forced to think harder about where the balls will drop.

In addition to the numbered keys (1–8), the plus (+) and minus (–) keys are active. Pressing plus drops a ball at random down one of the eight entry paths. Pressing minus allows you to pass your turn to your opponent.

Once you've mastered the regular game, you can add some new rules. Each player gets three passes per half, similar to the three timeouts in a football game. If you don't like the looks of the board, press the minus key to use one of your passes. After one player has skipped a turn, the other player must play (this prevents the possibility of six passes in a row). It's also a good idea to make the rule that a player can't pass on two consecutive turns. You can also give each player two random moves to be played for the opponent. In other words, after making a move, you would inform your opponent that you're going to give him or her one of your random moves. Then you'd press the plus key.

Here's one more possible change: Instead of alternating turns, allow a player to continue after scoring. When a player drops a ball and scores points, the other player will have to pass (by pressing the minus key). If the first player scores again, the opponent passes again and continues to pass until no more points are scored.

### **Solitaire**

If you'd like to play alone against the computer, you can use the pass and random turn options. To drop a ball, press a number key (1–8). The numeric keypad is convenient for choosing a move. Here are the rules for solitaire play:

1. The computer always scores first. At the beginning of each round, the computer plays randomly until at least one point is acquired. Press the plus key for the computer's turn. You must continue passing (skip your turn with the minus key) until the computer puts points on the board.



2. After the first score by the computer, you can begin to play. When the computer has a turn, press the plus key for a random move.
3. Whenever you make points, you must pass again until the computer scores. When the computer gets more points, you can begin to play again. This rule means that you should hold back on the easy scores of a few points. Wait until there's an avalanche coming.
4. If you're the first to reach the goal, the computer gets a last chance. Don't make this move randomly. Figure out the best opportunity for scoring, and play that move for the last-chance turn.

In the interest of keeping Switchbox a program of manageable length, no attempt has been made to provide an "intelligent" computer opponent. Once you're familiar with the game, you might find it an interesting project to add some routines that give the computer a rational basis for picking one move over another.

## Switchbox

```

10  restore: DIM SW(4,7,1),SP$(1),LB(32,4),AR$(1)
    ,PT(4,16),SC(1,8):qr=1
20  sp$(0)="\\_":SP$(1)="_//":AR$(0)=chr$(4)+" "
    :AR$(1)="" +chr$(3)
30  COLOR 1,1,1,1,1:q1=-2:q2=0:for j=1 TO 4:READ
    PT(J,0)
40  for a=0 to 1:for b=0 to 8:sc(a,b)=0:next:next
    t
50  FOR K=1 TO 7:READ L:PT(J,K+7)=L:PT(J,8-K)=L:
    NEXT K,J
60  DATA 10
70  DATA 2,2,2,2,2,2,2
80  DATA 40
90  DATA 1,2,3,5,8,13,21
100 DATA 20
110 DATA 2,3,4,5,6,7,8
120 DATA 80
130 DATA 1,4,9,16,25,36,49
140 fullw 2:clearw 2:GOTOXY 0,0:INPUT "PLAYER 1"
    ;P1$
150 INPUT "PLAYER 2";P2$:P1$=LEFT$(P1$,5):P2$=LE
    FT$(P2$,5):PRINT P1$;" VS ";P2$
160 PRINT "IS THIS CORRECT?":gk=inp(2):if gk<>as
    c("Y") and gk<>asc("y") then 140
170 GOSUB 410:GOSUB 510:COLOR 1,1,1

```

## CHAPTER TWO

```
180  FOR RR=1 TO 4:COLOR 12:gotoxy 0,1+rr:print "
    *";:gotoxy 28,1+rr:print "*"
190  GOSUB 450:REM PUT SCORES AT BOTTOM
200  QR=1-QR:TY=QR*20:TX=26-TY:CX=TX:CY=0
210  COLOR 2:M$=RIGHT$(STR$(PT(RR,0)),2):GOSUB 11
    10
220  cx=6+TY:CY=0:M$=AR$(QR):GOSUB 1110
230  GOSUB 660:IF SC(1-QR,RR) >= PT(RR,0) THEN 25
    0:REM END OF ROUND
240  GOTO 200
250  FOR J=0 TO 1:FOR K=5 TO 8:SC(J,K)=0:NEXT K,J
260  FOR J=0 TO 1:FOR K=1 TO 4:GL=PT(K,0):AC=SC(J
    ,K):SC(J,5)=SC(J,5)+AC
270  SC(J,6)=SC(J,6)-(AC>=GL)*GL:SC(J,7)=SC(J,7)+
    (SC(J,K)-SC(1-J,K)):NEXT K,J
280  FOR J=0 TO 1:FOR K=6 TO 7:SC(J,K)=SC(J,K)+SC
    (J,5):NEXT K,J
290  FOR J=0 TO 1:FOR K=5 TO 7:SC(J,8)=SC(J,8)+SC
    (J,K):NEXT K,J
300  FOR J=0 TO 1:FOR K=5 TO 8:Y$=STR$(SC(J,K)):L
    =LEN(Y$):TX=5+J*28-L
310  TY=2+K:CX=TX:CY=TY:M$=Y$:COLOR 4:GOSUB 1110:
    NEXT K,J
320  NEXT RR:REM END OF MAIN LOOP
330  COLOR 1,1,8
340  GOTOXY 9,10:PRINT SPC (19)
350  GOTOXY 9,11:PRINT " PLAY AGAIN? (Y/N) "
360  GOTOXY 9,12:PRINT SPC (19)
370  FOR A=78 TO 231 STEP 153:LINEF A,100,A,109:N
    EXT
380  LINEF 78,100,231,100:LINEF 78,109,231,109
390  A=INP(2):IF A=ASC("Y") OR A=ASC("y") THEN CL
    EAR:GOTO 10 ELSE END
410  clearw 2:COLOR 4,1,7
420  FOR J=1 TO 8:GOTOXY 7+2*J,0:PRINT J:NEXT
430  FOR J=82 TO 227 STEP 18.125:LINEF J,0,J,190:
    NEXT
440  LINEF 82,9,227,9:RETURN
450  FOR J=1 TO 14:K=PT(RR,J):JJ=2+J*2
460  IF K>9 THEN L=INT(K/10):L$=MID$(STR$(L),2,1)
    :GOTO 480
470  L$=CHR$(32)
480  gotoxy jj,16:print l$;:CX=JJ:CY=17
490  gotoxy jj,17:print right$(str$(k),1);
500  NEXT J:RETURN
510  GOSUB 580:FOR J=0 TO 3:SY=4+J*4:FOR K=0 TO J
    +3:SX=12-J*2+K*4
520  CX=SX-1:CY=SY-2:M$=" ":GOSUB 1110:COLOR 0,0,
    0,0,0
530  LINEF CX*9,CY*9+10,CX*9+3,CY*9+10:WP=INT(RND
    (1)*2)
```



```

540 SW(J,K,0)=WP:SW(J,K,1)=0:GOSUB 650
550 NEXT K,J:COLOR 9
560 CX=1:CY=0:M$=P1$:GOSUB 1110
570 CX=29:CY=0:M$=P2$:GOSUB 1110:RETURN
580 LINEF 82,51,64,69:LINEF 64,69,64,172
590 LINEF 64,87,46,105:LINEF 46,105,46,172
600 LINEF 46,123,28,141:LINEF 28,141,28,172
610 LINEF 228,51,246,69:LINEF 246,69,246,172
620 LINEF 246,87,264,105:LINEF 264,105,264,172
630 LINEF 264,123,282,141:LINEF 282,141,282,172
640 RETURN
650 COLOR 2: CX= SX-2: CY= SY-1: M$= SP$(WP): GOSUB 111
0: RETURN
660 FOR J=0 TO 32: LB(J,0)=0: NEXT: NB=1
670 A=inp(2): a$=chr$(a)
680 IF A$="-" THEN RETURN
690 IF A$="+" THEN A$=STR$(INT(RND(1)*8+1))
700 A=VAL(A$): IF (A<1) OR (A>8) THEN 670
710 LB(0,0)=1: FOR J=1 TO 3: LB(0,J)=0: NEXT: LB(0,4
)=10+A*2
720 EX=1
730 FOR J=0 TO 32: IF LB(J,0) THEN EX=0: GOSUB 760
740 NEXT: sound 1,0,0,0: IF EX THEN RETURN
750 GOTO 720
760 DY=LB(J,0): DX=LB(J,1): LY=LB(J,2): NY=LB(J,3):
NX=LB(J,4): qt=ly*4+ny
770 IF (LY+NY) AND ly<4 THEN GOTOXY NX+Q1,LY*4+N
Y+Q2:PRINT " "
780 COLOR 3: LB(J,3)=(NY+1) AND 3: ON NY+1 GOTO 79
0,810,860,880
790 IF LY>3 THEN LB(J,0)=0: GOTO 950: REM SCORING
ROUTINE
800 gosub 1120: ON INT(RND(1)*3+1) GOTO 1000,1010
,1020
810 VX=0: GOSUB 940: IF SW(WY,WX,1)=0 OR (SW(WY,WX
,0)=SD)=0 THEN 840
820 vx=1-2*sd: lb(j,1)=vx: lb(j,3)=ny+1
830 LB(J,4)=NX+VX: gotoxy nx+q1+vx,ly*4+ny+q2-(qt
<15):print "o": goto 1050
840 IF SW(WY,WX,0)=SD THEN LB(J,0)=0: SW(WY,WX,1)
=1: gosub 1120: GOTO 1030
850 LB(J,3)=NY+1: gosub 1120: ON INT(RND(1)*3+1) G
OTO 1000,1010,1020
860 LB(J,1)=0: LB(J,4)=NX+DX: gotoxy nx+q1+dx,ly*4
+ny+q2-(qt<15):print "o"
870 goto 1060
880 if qt<15 then gosub 1120
890 LB(J,2)=LY+1: GOSUB 940: SW(WY,WX,0)=1-SW(WY,W
X,0)

```

## CHAPTER TWO

```
900 IF SW(WY,WX,1)=0 THEN 930
910 LB(NB,0)=1:LB(NB,1)=0:LB(NB,2)=LY:LB(NB,3)=0
   :LB(NB,4)=NX+2-SD*4:NB=NB+1
920 SW(WY,WX,1)=0:gotoxy nx+q1+2-sd*4,ly*4+ny+q2
   -1:print " ":gosub 1070
930 SX=12-WY*2+WX*4:SY=4+WY*4:WP=SW(WY,WX,0):GOS
   UB 650:GOTO 1050
940 WY=LY:JX=int(NX/2)+ly-6:WX=int(JX/2):SD=JX A
   ND 1:RETURN
950 SF=PT(RR,nx/2-2)
960 SG=SC(QR,RR)+SF
970 TX=3+29*QR+(SG>9)+(SG>99)+(SG>999)
980 tY=RR+1:A%=MID$(STR$(SG),2):COLOR 1
990 CX=TX:CY=TY:M%=A%:GOSUB 1110:SC(QR,RR)=SG:GO
   TO 1080
1000 sound 1,15,1,3:wave 1,1,12,90,0:return
1010 sound 1,15,1,4:wave 1,1,12,90,0:return
1020 sound 1,15,6,3:wave 1,1,12,90,0:return
1030 for a=12 to 1 step-2:sound 1,15,a,5:wave 1,1
   ,10,20
1040 next:return
1050 return
1060 wave 16,2,0,1000,3:return
1070 wave 16,2,0,18000,5:return
1080 for a=7 to 1 step-1:sound 1,15,1,a:wave 1,1,
   12,90,2:next
1090 sound 1,0,0,0:return
1100 REM CHAR COMMAND
1110 gotoxy cx,cy:print m%:return
1120 gotoxy nx+q1,ly*4+ny+q2-(qt<15):print "o":re
   turn
```



# Reversi

Kevin Mykytyn

Article by Philip I. Nelson

---

*Play this adaptation of a classic strategy game on any Atari ST system with a color monitor. You may play it with a friend or try to best the computer.*

“Reversi” is a fresh translation of a venerable game known by several different names. Since ancient times, strategists have delighted in this game’s simple, yet challenging premise. This version is written in ST BASIC and makes good use of the computer’s graphics capabilities.

## Getting Started

Type in the program and save a copy before you run it. You can play Reversi in either low resolution or medium resolution. However, the display looks best in low resolution. The playing field consists of an  $8 \times 8$  grid of 64 squares. One player’s pieces are black, and the other’s are white. If you play against the computer, you have the white pieces.

Each game begins with four pieces—two black and two white—placed symmetrically in the middle of the board (Figure 1). The players alternate turns by placing pieces on the board. Play proceeds until every square is filled or until neither player can make a move. In cases where it’s impossible to move, you must pass your turn.

## Playing and Scoring

The object of the game is to have more pieces on the board than your opponent does at the end of the game. To place a piece on the board, move the mouse pointer to the desired square and click the left button once. If the move is legal, a piece of your color appears in the designated square. The computer beeps if you attempt to make an illegal move.

To take a turn, you must place one of your pieces so that one or more of the opponent’s pieces will lie in a straight line between two of yours. When you enclose an opponent’s pieces in this way, the enclosed pieces will change from the opponent’s color to yours. Your score is equal to the number of

## CHAPTER TWO

Figure 1. Beginning Screen

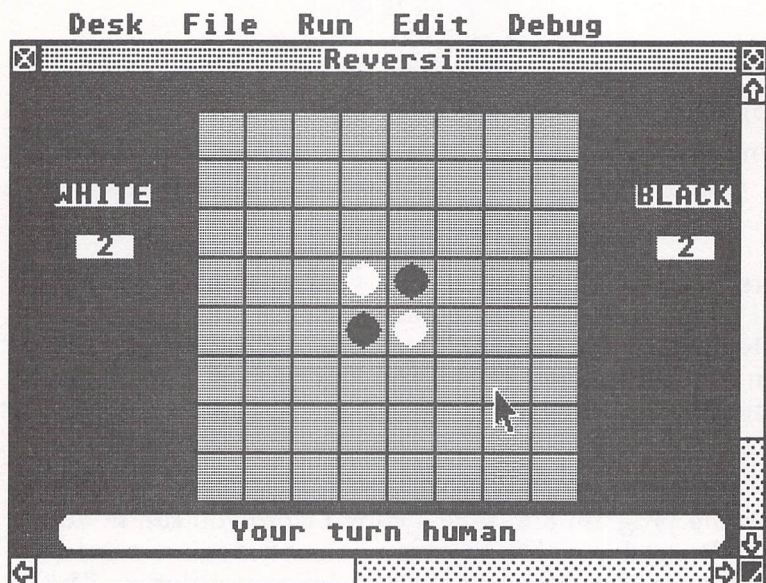


Figure 2. Before White's Move

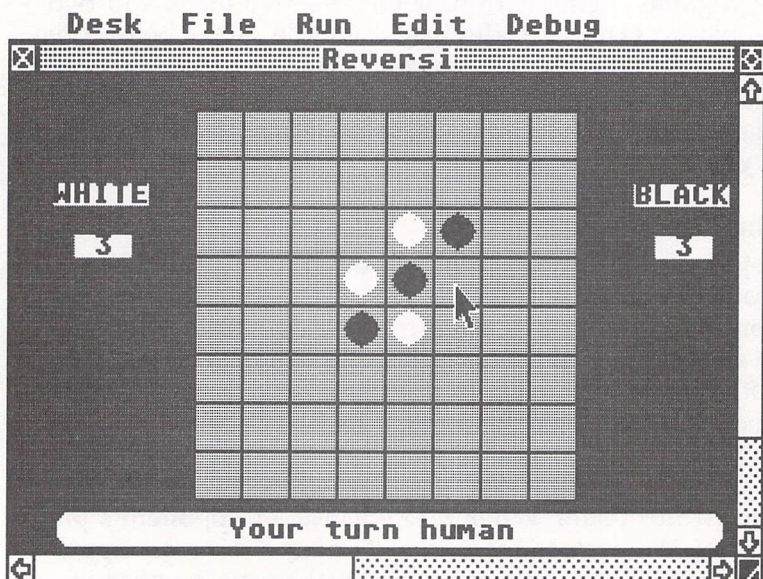
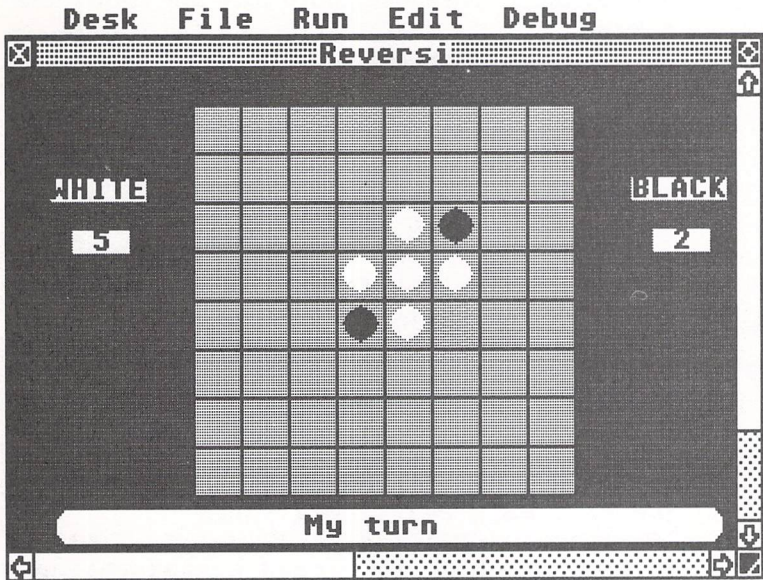




Figure 3. After White's Move



pieces you have on the board. The program displays both players' scores at all times and prompts you when it's time to make a move.

Figures 2 and 3 illustrate the effect of placing a piece on the board. In Figure 2, the white player is about to place a piece in the square indicated by the mouse pointer. Figure 3 shows the appearance of the board after that move is made.

## Two Levels

If you're playing against the computer, you may choose two different skill levels. Level 1 is easier than level 2, and it also plays faster. The higher level offers a greater challenge, but requires more time for the computer to calculate each move. Each of the computer's moves takes about 5 to 10 seconds at level 1 and about 20 to 50 seconds at level 2. Don't move the pointer while the computer is thinking; ST BASIC slows down when the pointer is in motion.

If you analyze the computer's strategy, you'll discover that it often tries to take the corner squares. The corners are the most valuable positions on the board because they can't be

## CHAPTER TWO

changed to the opposite color. Squares on the edge of the board are also strategically valuable, since they are vulnerable in only one direction.

Of course, there's no single strategy that works every time, particularly if you're playing a human opponent. Beginners often try to take the lead early and maintain it throughout the game, but that's not necessarily a winning strategy. When players are evenly matched, it's common for the score to seesaw back and forth several times. Dramatic reversals often occur near the end of the game—hence the name, Reversi. Experienced players try to think ahead and develop a strong strategic position with the final moves in mind.

### Reversi

```
10  dim board(9,9),tboard(8,8),dx(7),dy(7),path(
    7,1),mess$(2,1),sc1(9,9)
20  restore 40:for a=0 to 9:for b=0 to 9:board(a
    ,b)=4:next b,a
30  for a=0 to 7:read dx(a),dy(a):next
40  data 0,-1,1,-1,1,0,1,1,0,1,-1,1,-1,0,-1,-1
50  for a=1 to 2:for b=0 to 1:read mess$(a,b):ne
    xt b,a
60  data "          Your turn human          ","
    My turn                                "
70  data "          White's turn              ","
    Black's turn                          "
80  for a=1 to 4:for b=1 to 8:read c:sc1(a,b)=c:
    sc1(9-a,b)=c
90  next b,a
100 data 16,-4,4,2,2,4,-4,16,-4,-12,-2,-2,-2,-2,
    -12,-4
110 data 4,-2,4,2,1,4,-2,4,2,-2,2,0,0,2,-2,2
120 gosub SETSCREEN:p=0:gosub OPTIONS:nt=0
130 START: gosub SCORE
140 if np=2 or p=0 then 160
150 gosub TURN:gosub BESTMOVE:if js=-50 then 200
    else gosub CHECKLEGAL:goto 190
160 gosub ANYMOVE:if flag=0 then 200 else gosub
    TURN
170 gosub READMOUSE:gosub CHECKLEGAL
180 if flag=0 then gosub BEEP:goto 170
190 nd=0:gosub FLIPPIECES:nt=0
200 nt=nt+1:if nt=3 then goto GAMEOVER
210 p=1-p:goto START
220 SCORE: p1=0:p2=0:for a=1 to 8:for b=1 to 8
230 if board(a,b)=0 then p1=p1+1
240 if board(a,b)=1 then p2=p2+1
```



```

250 next b:next a:color 1,1,1:pt=p1+p2
260 gotoxy 2,4:print "WHITE":gotoxy 29,4:print "
    BLACK"
270 gotoxy 3,6:print p1:gotoxy 30,6:print p2
280 return
290 GAMEOVER: gosub SCORE:gotoxy 0,0:print:gotox
    y 4,17
300 if p1=p2 then print "It's a tie!";:goto 330
310 if p1>p2 then print "White wins!";:goto 330
320 if p2>p1 then print "Black wins!";
330 print " - Click mouse button";:gosub GETMOUS
    E
340 goto 20
350 TURN: color 1,1,1:gotoxy 0,0:print:gotoxy 4,
    17:print mess$(np,p):return
360 ANYMOVE: for tx=1 to 8:for ty=1 to 8
370 gosub CHECKLEGAL
380 if flag=1 then tx=9:ty=9
390 next ty,tx
400 return
410 CHECKMOVE: bs=-20:for tx=1 to 8:for ty=1 to
    8
420 gosub CHECKLEGAL:ns=sc1(tx,ty)
430 if flag=0 then goto 450
440 if ns>bs or ns=bs and rnd(1)>.5 then bs=ns
450 next ty,tx
460 return
470 BESTMOVE:js=-50:nd=1:for tx=1 to 8:for ty=1
    to 8
480 gosub CHECKLEGAL:if flag=0 then 560
490 for q=1 to 8:for r=1 to 8:tboard(q,r)=board(
    q,r):next q,r
500 gosub FLIPPIECES:fs=sc1(tx,ty):if pt>58 then
    fs=fs+f1*5
510 ptx=tx:pty=ty:if lev=2 then p=0:gosub CHECKM
    OVE:p=1
520 tx=ptx:ty=pty:for q=1 to 8:for r=1 to 8:boar
    d(q,r)=tboard(q,r):next r,q
530 if lev=1 then bs=0:goto 550
540 if pt>58 then bs=bs+f1*5
550 if fs-bs>js or (fs-bs>js and rnd(1)>.5) then
    js=fs-bs:gx=tx:gy=ty
560 next ty,tx:tx=gx:ty=gy
570 if (tx=1 or tx=8) and (ty=1 or ty=8) then fo
    r a=0 to 6 step 2:sc1(tx+dx(a),ty+dy(a))=8:n
    ext a
580 return
590 BEEP: sound 1,15,1,2,10:sound 1,0,0,0,0:retu
    rn
600 BONG: sound 1,15,8,3:wave 1,1,0,10000,10:ret
    urn

```

## CHAPTER TWO

```
610 GETMOUSE: poke contr1,124
620 poke contr1+2,0: poke contr1+6,0
630 vdisys(0)
640 mx=peek(ptsout):my=peek(ptsout+2)
650 if peek(intout)=0 then GETMOUSE
660 vdisys(0):if peek(intout)<>0 then 660
670 return
680 READMOUSE: gosub GETMOUSE
690 if mx<80 or mx>235 or my<35 or my>169 then R
EADMOUSE
700 tx=int((mx-80)/20)+1:ty=int((my-35)/17)+1
710 if board(tx,ty)<>4 then gosub BEEP:goto READ
MOUSE
720 return
730 FLIPPIECES: fl=0:x=tx:y=ty:gosub PUTPIECE
740 for a=0 to 7
750 if path(a,0)=0 then 800
760 x=tx+dx(a):y=ty+dy(a)
770 for b=1 to path(a,1)
780 gosub PUTPIECE:x=x+dx(a):y=y+dy(a)
790 next b
800 next a:return
810 CHECKLEGAL: q=1-p:flag=0:if board(tx,ty)<>4
then return
820 for a=0 to 7:path(a,0)=0
830 if board(tx+dx(a),ty+dy(a))<>q then 890
840 sx=tx+dx(a):sy=ty+dy(a):counter=0
850 checkpath: counter=counter+1:sx=sx+dx(a):sy=
sy+dy(a)
860 if board(sx,sy)=4 then 890
870 if board(sx,sy)=p then flag=1:path(a,0)=1:pa
th(a,1)=counter:goto 890
880 goto checkpath
890 next a:return
900 PUTPIECE: fl=fl+1:board(x,y)=p:if nd=1 then
return
910 PUTPIECE2: px=x*20+67:py=y*17+3
920 color p,p,p:pcircle px,py,7:gosub BONG
930 return
940 SETSCREEN: openw 2:fullw 2:clearw 2:title$="
Reversi":gosub SETTITLE
950 x1=20:y1=174:x2=300:y2=187:pi=8:gosub BOX
960 color 3,3,3:fill 100,100:color 1,1,1
970 for a=77 to 237 step 20:linef a,12,a,148:nex
t
980 for a=12 to 148 step 17:linef 77,a,237,a:nex
t
990 color 2,2,2:fill 20,20
1000 nd=0:for x=4 to 5:y=x:p=0:gosub PUTPIECE:nex
t
```



```
1010 p=1:x=4:y=5:gosub PUTPIECE:x=5:y=4:gosub PUT
    PIECE
1020 return
1030 SETTITLE: a# = gb : gintin = peek(a#+8)
1040 poke gintin+0,peek(systab+8) : poke gintin+2
    ,2
1050 s# = gintin+4 : title$ = title$ + chr$(0)
1060 poke s#,varptr(title$) : gemsys(105)
1070 return
1080 OPTIONS: a$="Number of players          1  2"
    :gosub MENU:np=ans
1090 if np=2 then return
1100 a$="Choose level (1 is easy)  1  2":gosub ME
    NU:lev=ans
1110 a$="Do you want to go first  Y  N":gosub ME
    NU:p=ans-1
1120 return
1130 MENU: gotoxy 0,0:print:gotoxy 4,17:print a$;
1140 gosub GETMOUSE:if my<175 or my>187 then 1140
1150 if mx>242 and mx<255 then ans=1:return
1160 if mx>264 and mx<280 then ans=2:return
1170 goto 1140
1180 BOX: poke contrl,11:poke contrl+2,2:poke con
    trl+6,0:poke contrl+10,pi
1190 poke ptsin,x1:poke ptsin+2,y1
1200 poke ptsin+4,x2:poke ptsin+6,y2
1210 vdisys(0):return
```

# 3-D Tic-Tac-Toe

David Bohlke

---

*This new rendition of an old favorite lets you match wits against the ST in a three-dimensional contest. If you like, you can even make changes to the program which will make the computer play more aggressively or more cautiously. Works with any Atari 520ST or 1040ST computer with a color monitor.*

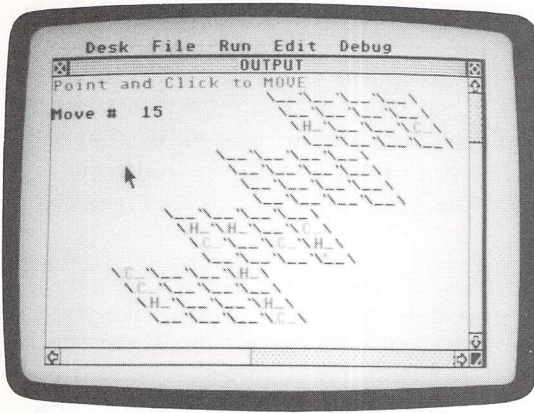
In "3-D Tic-Tac-Toe," you take on the Atari ST in a battle of wits. The object of this strategy game is similar to that of the traditional Tic-Tac-Toe game, except this version takes place in a simulated three-dimensional space which contains four game boards. To win, you must place four pieces in a row. The row may extend across a single plane or vertically through all four planes. Though it's not a flawless player, the ST is a formidable opponent.

## Entering Tic-Tac-Toe

Type in the program as listed and save it to disk. The program works in either low- or medium-resolution mode. When you run the program, it randomly selects you or the computer to go first. The computer needs only a few seconds to pick its move and places a red uppercase C at the selected square. The ST takes less time to move if you refrain from moving the mouse pointer around while it is calculating; moving the pointer freezes normal BASIC operations. You should also avoid moving the slider bars on the Output window, since this may jumble part of the game board.

It's your turn when the screen prompt appears. Use the mouse to move to the square of your choice; then click the left mouse button. Due to the slowness of ST BASIC, you may need to hold the button down for as long as one second before the computer recognizes your choice. A blue uppercase H appears on the square you have chosen. The H, of course, stands for the human—you—and the C stands for computer.





*"3-D Tic-Tac-Toe" challenges you to best the computer in a three-dimensional strategic simulation.*

## Program Strategy

You may be interested in learning how the ST plays this simple strategy game. The computer does not use a "look-ahead" technique; rather, it determines its move by assigning a numeric value to each empty square. Look at the table for an explanation of this value. It shows a sample Tic-Tac-Toe combination of four squares in a row, along with the corresponding BASIC line number that assigns the value.

## Combination Values

Line	Pattern	Value
540	HHHH	Human wins
540	CCCC	Computer wins
550	H_HH	33 points
560	_H_H	5 points
570	__H_	2 points
580	CC_C	77 points
590	C__C	6 points
600	_C__	1 point

Each computer piece is stored with a value of 5 in the V( ) array, and each human piece has a value of 1 in the array. So if a row of four squares contains two computer pieces, that combination places a value of 10 in the V( ) array. Line 530 evaluates the pieces in each row and assigns a combination value to the variable P. Lines 540–600 convert these combination values into point values, which are evaluated to choose the next move. Note that the order of pieces in the table has

no significance: What matters is the number of pieces and blanks. In the third entry, for instance, the sequence H\_HH merely indicates that the row contains one blank and three human pieces, in any order. No value is assigned to a row that contains both computer pieces and human pieces because it's clearly impossible for either opponent to win on that row.

### Program Modifications

This game is designed so that the computer plays a nearly equal balance of offense and defense. If you would like the computer to play more aggressively, increase the values for offensive moves in lines 590 and 600. For a more conservative game, you can increase the values in lines 560 and 570. With a little experience, you'll find that a change of just one or two points in these four lines will make a significant difference in the computer's move strategy.

### 3-D Tic-Tac-Toe

```
100 fullw 2:clearw 2
110 dim b(64),v(64),x(64),m(64,28):gosub 670
120 ' new game
130 clearw 2:color 1:print:for s=1 to 64:gosub 8
    70
140 gotoxy x-1,y:print"\_\_\";next
150 for i=1 to 64:b(i)=0:x(i)=0:v(i)=0:next:w(1)
    =0:mv=0
160 randomize 0:if rnd(1)<.5 then s=int(rnd(1)*6
    4)+1:gosub 840:color 2:goto 370
170 ' human moves
180 gosub 840:color 4:print:gotoxy 0,0:print"Poi
    nt and Click to MOVE"
190 gosub mousexy:mx=int(msx/9):my=int(msy/9.3)
200 sq=0:if msb<>1 then 190
210 for s=1 to 64:gosub 870
220 if y=my-2 and abs(x-mx)<=1 then sq=s
230 next:if sq=0 then 190
240 s=sq:gosub 870
250 if b(s)<>0 then 190
260 sx=1:gotoxy x,y:print"H_";b(s)=1:v(s)=0:gos
    ub 520
270 if w(1)>0 then 440
280 ' computer moves
290 gosub 840:color 2:print:gotoxy 0,0:print"Ata
    ri ST's Move"
300 sx=0:for s=1 to 64:if b(s)>0 or x(s)=0 then
    310 else v(s)=0:gosub 520
```



```

310 next
320 s=0:h=0:for i=1 to 64
330 if v(i)=h and rnd(1)<.3 and h>0 then h=v(i):
    s=i
340 if v(i)>h then h=v(i):s=i
350 next
360 if s=0 then gotoxy 0,0:print"      DRAW game
    ";:a$="D":color 1:w(1)=1: w(2)=2:w(3)=
3: w(4)=4:goto 460
370 gosub 870:b(s)=5:v(s)=0
380 for i=1 to 4:gotoxy x,y:print" *";:sound 1,8
    ,1,4,10
390 gotoxy x,y:print"C_";:sound 1,8,1,5,10:next:
    sound 1,0,0,0,0
400 sx=1:for i=1 to 64:x(i)=0:next:gosub 520
410 if w(1)>0 then 450
420 goto 170
430 ' game over
440 gotoxy 0,0:print"You WIN
    ";:a$="H":goto 460
450 gotoxy 0,0:print"Computer WINS      ";:a$="C"
460 gotoxy 0,1:print"CLICK for new game";
470 for i=1 to 4:s=w(i):gosub 870:gotoxy x,y:pri
    nt a$;:next:for i=1 to 99:next
480 sound 1,8,5,5,10:sound 1,0,0,0,0
490 for i=1 to 4:s=w(i):gosub 870:gotoxy x,y:pri
    nt" ";:next:for i=1 to 99:next
500 gosub mousexy:if msb<>0 then 120 else 470
510 ' adjust value array V(64) for computer move
    at square s
520 eg=0:j=1:for i=1 to m(s,0)
530 p=0:for k=1 to 4:p=p+b(m(s,j)):j=j+1:next:q=
    0
540 if p=4 or p=20 then for k=0 to 3:w(k+1)=m(s,
    j+k-4):next
550 if p=3 then q=33:goto 620
560 if p=2 then q=5:goto 620
570 if p=1 then q=2:goto 620
580 if p=15 then q=77:goto 620
590 if p=10 then q=6:goto 620
600 if p=5 then q=1:goto 620
610 if sx=1 then 620 else 660
620 v(s)=v(s)+q:if b(s)>0 then v(s)=0
630 if sx=0 then 660
640 for k=0 to 3:if b(m(s,j+k-4))=0 then x(m(s,j
    +k-4))=1
650 next
660 next:return
670 ' load legal win combos into M(64,28)
680 clearw 2:color 1:print" Loading DATA ..."
```

## CHAPTER TWO

```
690 for i=1 to 64:m(i,0)=0:next
700 for i=1 to 16:a=i*4-3:for j=1 to 4:w(j)=a:a=
a+1:next:gosub 820:next
710 for i=1 to 4:for j=i to i+48 step 16:n=j
720 for k=1 to 4:w(k)=n:n=n+4:next:gosub 820:nex
t:next
730 for i=1 to 16:for j=0 to 3:w(j+1)=j*16+i:nex
t:gosub 820:next
740 for i=1 to 28:for j=1 to 4:read a:w(j)=a:nex
t:gosub 820:next:return
750 data 1,21,41,61,2,22,42,62,3,23,43,63,4,24,4
4,64
760 data 1,18,35,52,5,22,39,56,9,26,43,60,13,30,
47,64
770 data 4,19,34,49,8,23,38,53,12,27,42,57,16,31
,46,61
780 data 13,25,37,49,14,26,38,50,15,27,39,51,16,
28,40,52
790 data 1,6,11,16,17,22,27,32,33,38,43,48,49,54
,59,64
800 data 4,7,10,13,20,23,26,29,36,39,42,45,52,55
,58,61
810 data 1,22,43,64,4,23,42,61,13,26,39,52,16,27
,38,49
820 for k=1 to 4:l=m(w(k),0)*4+1:m(w(k),0)=m(w(k
),0)+1
830 for p=1 to 4:m(w(k),1)=w(p):l=l+1:next:next:
return
840 color 1:mv=mv+1:gosub clrprt:gotoxy 0,2:prin
t"Move # ";mv;:return
850 clrprt:gotoxy 0,0:print spc(23);:return
860 ' input s=square to move to, returns x,y as
print position
870 a=int((s-1)/16):y=a*4+3:b=s-a*16
880 c=int((b-1)/4):y=y+c-2:x=(4-a)*4+c
890 x=x+(b-c*4)*3-1:return
900 mousexy:poke contrl,124:poke contrl+2,0
910 poke contrl+6,0:vdisys(0)
920 msx=peek(ptsout):msy=peek(ptsout+2):msb=peek
(intout):return
```



CHAPTER THREE

**Applications  
and Education**





# Hickory, Dickory, Dock

Barbara H. Schulak  
Version by Kevin Mykytyn and David Florance

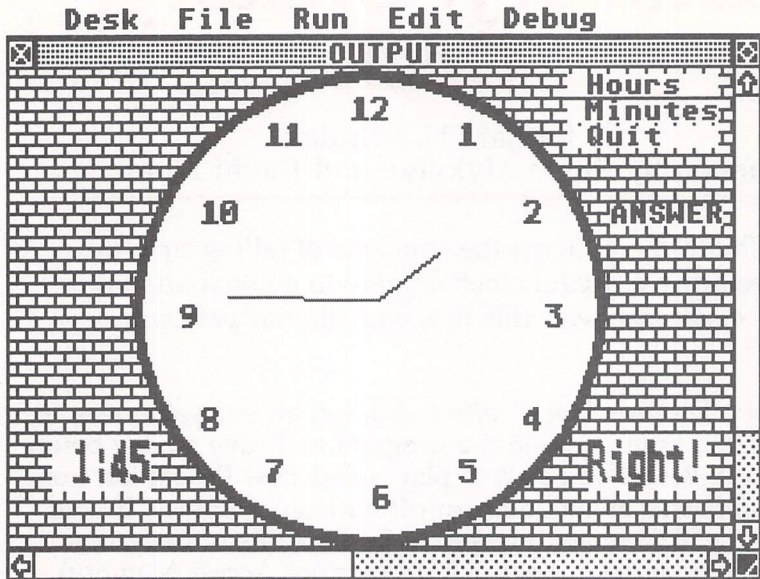
---

*Children can learn the concepts of telling time by relating a digital clock display to a conventional clock face with this fun, educational program.*

"Hickory, Dickory, Dock" offers children an enjoyable way to learn to tell time. Type in the program and save a copy before you run it. Before you start to play, select Low Resolution from the Preferences menu and turn off buffered graphics. If your ST has the TOS operating system in ROM (Read Only Memory), there may be enough RAM (Random Access Memory) left to run the program without turning off buffered graphics.

When you run Hickory, Dickory, Dock, the screen will list five options. Enter a number, 1-5, to choose the option you want. Four different activities are available. The fourth option, Practice, lets children practice moving the clock hands before they start the actual testing. In this option, as the positions of the clock hands change on the screen, a digital clock display changes as well. The child can thus see the relationship between the spatial position of hands on a clock face and the numeric representation of time.

The other three activities test a child's time-telling ability for hours only, for hours and half-hours, and for five-minute intervals. When one of these options is chosen, a round clock face and a digital time display will appear on the screen. The object is for the child to set the clock hands to match the time shown in the digital display box. The child uses the mouse to move the hands and then to enter the answer. After five correct answers have been given, a brief song is played as a reward. If three incorrect choices are made, the program automatically moves the clock hands to the correct position.



*"Hickory, Dickory, Dock" offers children a fun way to learn about telling time.*

### How to Play

Playing is simple and requires only the use of the number keys (1–5) and the mouse. Beside the clock, in the box in the lower left corner, you will see a time given in digital form. Using the left mouse button, move the hands on the clock face to match this time. Just follow these steps:

- Click with the left mouse button on *Hours* to move the blue hours hand.
- Click *Minutes* to move the red minutes hand.
- When you have set the hands to match the time, click *Answer* to enter your answer.
- The ST will let you know if your answer is right or wrong. If you enter three wrong answers, it will move the hands to the correct positions for you. If you get five correct answers in a row, you'll be rewarded with a song.
- Click *Quit* when you want to return to the main menu.



## Hickory, Dickory, Dock

```
10  randomize 0:goto 30
20  z=a*.0175:x=int(xc-xr*cos(z)):y=int(yc-yr*sin(z)):return
30  ch=14:gosub 470:openw 2:fullw 2:clearw 2:ohx
    =130:omx=130:ohy=75:omy=75
35  color 1,1,1,1,1
40  nr=0:xr=70:yr=60:xc=130:yc=75:gotoxy 7,1:pr
    nt "Hickory, Dickory, Dock"
42  gotoxy 11,4:print "COMMAND MENU"
45  gotoxy 0,7:print "    1. Test - hours"
50  print:print "    2. Test - hours and half hou
    rs"
60  print:print "    3. Test - five minute interv
    als"
70  print:print "    4. Practice":print:print "
    5. Quit"
80  k=inp(2):if k<49 or k>53 then 80
90  if k=53 then ch=6:gosub 470:end else if k=5
    2 then 250
100 n=k-48:gosub 280
110 hr=int(rnd(1)*12)+1:mn=0:if n=2 then mn=int(
    rnd(1)+.5)*30:goto 130
120 if n=3 then mn=int(rnd(1)*12)*5
130 gosub 380:amn=mn:ahr=hr:mn=0:hr=12
140 gosub 340:gosub 420:if k=113 then 30
150 if k=13 then 160
155 goto 140
160 color 2:if amn<>mn or ahr<>hr then 190
170 nr=nr+1:gotoxy 27,15:print "Right!":b=1:c=5:
    gosub 560
180 nw=0:gosub 240:goto 220
190 gotoxy 27,15:print "Wrong!":b=5:c=1:gosub 56
    0
200 gosub 240:hr=ahr:mn=amn:nw=nw+1:if nw<>3 the
    n 130
210 gosub 340:for td=1 to 4000:next:nw=0:goto 23
    0
220 if nr=5 then nr=0:gosub 500
230 for t=1 to 500:next:goto 110
240 for td=1 to 300:next:sound 1,0:gotoxy 27,15:
    print "    ":return
250 gosub 280
260 gosub 340:gosub 380:gosub 420:if k=113 then
    30 else 260
270 ' draw the clock
280 clearw 2:color 1,5,1,1,1
281 ellipse 151,83,99,83:ellipse 151,83,96,80:fi
    ll 79,30
290 ch=6:gosub 470
```

## CHAPTER THREE

```
300 color 6,4,4,9,2:fill 10,10:fill 200,10
305 color 1:ch=7:gosub 470:nr=0:nw=0:for q=1 to
    12:a=q*30+90:gosub 20
310 x=int(x/8):y=int(y/8):if q=11 or q=12 then x
    =x-1
320 gotoxy x,y:print q:next
322 ch=6:gosub 470:gotoxy 26,0:print " Hours "
324 gotoxy 26,1:print " Minutes"
326 gotoxy 26,2:print " Quit "
328 gotoxy 28,5:print "ANSWER";
330 xr=xr-7:yr=yr-7:xc=xc+25:yc=yc+5:mn=0:hr=12:
    return
340 color 1,1,0:linef xc,yc,omx,omy:linef xc,yc,
    ohx,ohy
350 color 1,1,2:a=mn*6+90:gosub 20:linef xc,yc,x
    ,y:xr=xr/2:yr=yr/2
360 color 1,1,4:omx=x:omy=y:a=hr*30+90+mn/2:gosu
    b 20:linef xc,yc,x,y
370 xr=xr*2:yr=yr*2:ohx=x:ohy=y:return
380 ch=16:gosub 470
385 color 2:gotoxy 2,15:if hr<10 then print " ";
390 q=hr:gosub 410:ch=16:gosub 470:print q$;" ";
    :if mn<10 then print "0";
400 q=mn:gosub 410:ch=16:gosub 470:print q$:gosu
    b 240:return
410 q$=right$(str$(q),len(str$(q))-1):return
420 k=0:gosub 600:if button=0 then 420
422 if x<226 or x>300 then 420
425 if y<37 and y>31 then mn=mn+5:if mn=60 then
    mn=0
430 if y<30 and y>24 then hr=hr+1
440 if hr=13 then hr=1
450 if y<48 and y>42 then k=113:goto 465
452 if y<74 and y>68 then k=13:goto 465
455 if y<37 and y>24 then 465
460 goto 420
465 return
470 poke contrl,12:poke contrl+2,1:poke contrl+6
    ,0
480 poke ptsin,0:poke ptsin+2,ch
490 vdisys (0):return
500 restore 520:for nt=1 to 27:read a,b,c
510 sound 1,15,a,b,c*7:sound 1,0:next:sound 1,0:
    return
520 data 8,5,1,8,5,1,8,5,1,10,5,1,10,5,1,10,5,1,
    8,5,5
530 data 8,5,1,5,5,2,5,5,1,6,5,2,6,5,1,5,5,5
540 data 5,5,1,3,5,2,3,5,1,10,5,3,8,5,2,8,5,1,1,
    6,3
```



```
550 data 8,5,1,10,5,1,8,5,1,6,5,1,5,5,1,3,5,1,1,  
5,6  
560 for a=b to c step 2*sgn(c-b):sound 1,15,a,6  
570 wave 1,1,14,5,5:next:return  
600 poke contr1,124  
610 poke contr1+2,0:poke contr1+6,0  
620 vdisys(0)  
630 x=peek(ptsout):y=peek(ptsout+2)  
640 button = peek(intout)  
650 return
```

# Multiple-Choice Test Generator

C. Regena

---

*Generating test questions is a popular computer application because a computer can mix up the order of questions so that each run is different. "Test Generator" allows you to create questions for a multiple-choice test. You can take the test onscreen or print copies on your printer.*

Multiple-choice tests are handy for students to use in their schoolwork or for adults to brush up on subjects of interest to them. "Test Generator" gives you a way to make up your own multiple-choice tests. You can save the questions as a disk file and load the questions in later. You can print a copy of a test to be performed on paper, or you can use the computer to take the test. If you perform the test on the computer, you can also print that test with your answers and score on the printer.

To generate the multiple-choice test, the computer randomly chooses questions from all the possible questions. No question is repeated. Each question's answers are also printed in a random order. Therefore, each time you run the program, the questions appear in a different order and the answers for each question may be in a different order. If you want to print ten copies of the test, for example, you can get ten different variations of the test.

## Devising and Editing Test Questions

Test Generator is designed to run in medium resolution on the 80-column screen. Use the Set Preferences option to select medium resolution.

The first thing you need to do is create questions for your test. Run the program; then choose option 1, CREATE NEW TEST. You will be asked to enter a filename for this set of test questions. Enter a filename of up to eight characters—for example, GEOMETRY or HISTORY. Now start entering questions and answers at the prompts. First, enter a question and



press either Return or the Enter key. Next enter the four possible answers, one at a time. Finally, enter the number of the correct answer as you have entered it. For example, 3 indicates that the third answer is the correct answer.

After you've entered a question and its answers, you may edit the question as necessary by pressing function keys. To save the question, press F1. When you press F1 for the first question to be saved, you will hear the disk drive. However, after subsequent questions, the disk will not operate until several questions have been entered. To stop entering questions, enter a zero or simply press Return or Enter. After the disk drive has saved information, you will return to the main menu screen.

When your test questions are in memory, you may edit the questions, save them again for a backup copy, print a test, or perform a test. Once you have created a test (option 1), a copy of the questions is saved on disk. You may load them by using option 2 when you want to run the program later. Before you can perform the other options, you must either create a test or load a test.

When you edit the questions in memory, each question in turn appears on the screen with its possible answers and the correct answer. A list of the function keys and editing options is displayed. Pressing F1 saves the question and its possible answers as they are. F2 allows you to change the question. F3 shows the four answers, and you may change any or all of them. If you want to keep the question or answers as they are, simply press Return or Enter. Press F4 to change the correct answer. Whenever you are asked to enter the correct answer, you will need to input a number from 1 to 4. Press F10 to delete the question altogether.

Lines 290-640 contain the subroutine for editing. This subroutine is called after each question is entered when you create a test or when you edit the questions in memory. Lines 650-710 form a subroutine that enters the number of the correct answer.

Option 4 lets you save the questions in memory. You may use it to save a backup copy of the questions or to save test questions on different disks. Simply enter a filename for the test, and the computer will save all the questions currently in memory.

### Taking a Test

The fifth option is to print a hardcopy test on the printer. You may choose any number of questions to be used in the test, up to the maximum number of questions in memory. For example, if you have created a test of 30 questions, you may choose to print a test containing only 10 of those questions. LPRINT is the command used to print something on the printer. If you have long printed lines, you may need to adjust the printing (by specifying a width or adding spaces) so that words are not split on the lines. The procedure for printing a test uses the same code as a test that is being performed, but it doesn't print the answer chosen and tell whether it is correct or not. After the test questions are printed, an answer key is printed as well.

Use option 6 when you want to perform the test on the computer. Again, you may choose the number of questions you want for your test. A random question is printed on the screen with its four possible answers. You need to press 1, 2, 3, or 4 to indicate the correct answer. If your answer is incorrect, the correct answer is given. You may also choose to print this test on the printer as you are performing it. The printed copy shows the questions with their possible answers and the answers you entered.

### How the Program Works

Sequential files are used to save and retrieve information. The command

**OPEN "O,"#2,F\$**

opens the channel for output or prepares a disk file to save information. The letter *O* enclosed in quotation marks stands for *output*. I chose to use #2 for any output files. F\$ is the filename which you will be asked to enter when you run the program.

PRINT #2 prints the information to the disk file. Sometimes when you print a list of items, you need to specify the delimiters between items. In this program it is necessary to put commas in the list. Line 410 saves the question T\$, the four possible answers in AB\$, and the correct answer B. The string "ZZZ" is used at the end of the questions, and the file is closed with CLOSE #2 (lines 1090–1100, 1490–1500, and 1560).



To load a file or to retrieve the information, the statement is

**OPEN "I,"#3,F\$**

This statement opens channel 3 for *input*. The filename is F\$, which must be a filename for a test that has previously been saved. INPUT #3 is then used to read the information in the same order in which it was saved. Again, "ZZZ" is used to check for the end of input. EOF (end-of-file) is another method you may prefer to use.

**Note:** ST BASIC seems to have some problems with file processing. A problem may occur when you run the program and load files repeatedly. Extra characters may appear or the information may get mixed up. Due to the way Atari BASIC stores variables, you may find it necessary to reload BASIC and the program after the latter has saved or loaded data from disk the first time.

The variable N is used to indicate the number of questions that are available. When printing or performing the test, NT is the number of questions you want in the test. The S\$ array is used to prevent repetition of questions. The original array contains "A" for each element. As a question is chosen, S\$ is changed to "B" so that it will not be chosen again. Lines 1840–1860 randomly choose the question.

Lines 1910–2040 randomly arrange the four answers and print them. The C array makes sure that answers are not repeated and that all four answers are used. D is the random number chosen which will be the correct answer. ANS keeps track of the answers for the answer key which is printed when a test is printed. AA\$ is the array used to arrange the four answers.

INP(2) is used rather than INPUT where only one key needs to be pressed—for example, to enter the answer to the question. E=INP(2) returns the ASCII code of the key pressed. If the key pressed is a number from 1 to 4, E must be 49, 50, 51, or 52. If you subtract 48, E will be the actual number pressed. E and EE are used as choices made, and K is used for the answer chosen. The score is SC, and it is incremented for each correct answer.

## CHAPTER THREE

### Test Generator

```
10 REM TEST GENERATOR
20 CLEAR:DEFINT N,B,E,J,K,S
30 DIM T$(50),AB$(50),B(50),S$(50),A$(4),AA$(4)
   ,ANS(50)
40 FULLW 2:CLEARW 2:WIDTH 76
50 GOTOXY 4,1
60 PRINT "*** MULTIPLE CHOICE TEST ***"
70 PRINT:PRINT
80 PRINT "CHOOSE:"
90 PRINT " 1 CREATE NEW TEST"
100 PRINT " 2 LOAD TEST"
110 PRINT " 3 EDIT QUESTIONS IN MEMORY"
120 PRINT " 4 SAVE BACK-UP COPY OF QUESTIONS IN
   MEMORY"
130 PRINT " 5 PRINT (HARDCOPY) TEST"
140 PRINT " 6 PERFORM TEST ON COMPUTER"
150 PRINT " 7 END PROGRAM"
160 E=INP(2)
170 IF E<49 OR E>55 THEN 160
180 EE=E-48:CLEARW 2:GOTOXY 0,0
190 ON EE GOTO 720,1120,1240,1520,1580,1590,2370
200 REM GET 4 ANSWERS
210 L=1:AD$=AB$(NN)
220 FOR J=1 TO 3
230 A=INSTR(L,AD$,"/")
240 A$(J)=MID$(AD$,L,A-L)
250 L=A+1
260 NEXT J
270 A$(4)=RIGHT$(AD$,LEN(AD$)-A)
280 RETURN
290 REM EDITING
300 ??:? "CHOOSE:"
310 PRINT "F1 --EVERYTHING IS CORRECT; SAVE"
320 PRINT "F2 --CHANGE QUESTION"
330 PRINT "F3 --CHANGE POSSIBLE ANSWERS"
340 PRINT "F4 --CHANGE CORRECT ANSWER"
350 PRINT "F10--DELETE THIS QUESTION"
360 E=INP(2):IF E=196 THEN 640
370 IF E<187 OR E>190 THEN 370
380 ON E-186 GOTO 390,430,490,490
390 PRINT:PRINT "...SAVING..."
400 AB$(K)=A$(1)+"/"+A$(2)+"/"+A$(3)+"/"+A$(4)
410 PRINT #2,T$(K);", ";AB$(K);", ";B(K)
420 GOTO 640
430 PRINT:PRINT "QUESTION";K
440 PRINT:PRINT T$(K):PRINT
450 PRINT "ENTER REVISED QUESTION, OR JUST PRESS
   <RETURN> FOR NO CHANGES."
460 PRINT:INPUT D$
```



```
470 IF LEN(D$)<>0 THEN T$(K)=D$
480 GOTO 300
490 PRINT:PRINT
500 FOR J=1 TO 4
510 PRINT J;A$(J)
520 NEXT J:PRINT
530 IF E=190 THEN 620
540 PRINT:PRINT "Press <RETURN> to keep the answer"
550 PRINT "or enter new possible answer."
560 FOR J=1 TO 4
570 PRINT:PRINT J;A$(J)
580 INPUT D$
590 IF LEN(D$)=0 THEN 610
600 A$(J)=D$
610 NEXT J
620 GOSUB 650
630 GOTO 300
640 RETURN
650 PRINT:PRINT "ENTER THE CORRECT ANSWER--1, 2, 3, OR 4";
660 INPUT BB:BB=INT(BB)
670 IF BB>0 AND BB<5 THEN 700
680 PRINT "PLEASE ENTER A NUMBER FROM 1 TO 4."
690 GOTO 650
700 B(K)=BB
710 RETURN
720 PRINT "CREATE A MULTIPLE CHOICE TEST"
730 ?:?
740 INPUT "ENTER NAME OF TEST FILE: ";F$
750 IF LEN(F$)<>0 THEN 780
760 ?:"Please enter a name of up to 8 characters."
770 GOTO 730
780 F$=LEFT$(F$,8):N=0
790 OPEN "O",#2,F$
800 ?:"You may enter first a question, then four possible answers."
810 ?:"Next indicate the correct answer."
820 ?:"To stop entering questions, enter 0 or just press RETURN or ENTER."
830 N=N+1:K=N:IF N<51 THEN 870
840 ?:"This test is designed for up to 50 questions only."
850 ?:"You have entered 50 questions."
860 GOTO 1090
870 ?::?"ENTER QUESTION";N
880 INPUT T$(N)
890 IF T$(N)="" OR LEN(T$(N))=0 THEN 1020
900 PRINT:PRINT
```

## CHAPTER THREE

```
910 PRINT "NOW ENTER POSSIBLE ANSWERS."
920 FOR J=1 TO 4
930 PRINT:PRINT "ANSWER";J;": ";
940 INPUT A$(J)
950 IF LEN(A$(J))<>0 THEN 980
960 PRINT "PLEASE ENTER AN ANSWER."
970 GOTO 930
980 NEXT J
990 GOSUB 650
1000 GOSUB 300
1010 IF E=196 THEN 870 ELSE 830
1020 N=N-1
1030 ?:"You have entered";N;"questions."
1040 ?:"CHOOSE:"
1050 ?"F1--ENTER MORE QUESTIONS"
1060 ?"F2--STOP ENTERING QUESTIONS"
1070 E=INP(2):IF E=187 THEN 830
1080 IF E<>188 THEN 1070
1090 PRINT #2,"ZZZ"
1100 CLOSE #2
1110 GOTO 70
1120 PRINT "LOAD A PREVIOUSLY CREATED TEST"
1130 ?:"?:" "ENTER NAME OF TEST."
1140 INPUT F$
1150 IF LEN(F$)=0 THEN 1130
1160 F$=LEFT$(F$,8)
1170 OPEN "I",#3,F$
1180 CLOSE #3:OPEN "I",#3,F$
1190 N=1
1200 INPUT #3,T$(N)
1210 IF T$(N)="ZZZ" THEN N=N-1:CLOSE #3:GOTO 40
1220 INPUT #3,AB$(N),B(N)
1230 PRINT N:N=N+1:GOTO 1200
1240 PRINT "EDIT THE TEST IN MEMORY"
1250 IF N<>0 THEN 1290
1260 ?:"You need to create a test or load a test
"
1270 ?"before you can edit or save questions."
1280 GOTO 70
1290 PRINT:PRINT
1300 INPUT "ENTER NAME OF NEW TEST FILE: ";F$
1310 IF LEN(F$)<>0 THEN 1340
1320 PRINT "Please enter a file name of up to 8 c
haracters."
1330 GOTO 1290
1340 F$=LEFT$(F$,8)
1350 OPEN "O",#2,F$
1360 IF EE=4 THEN 1530
1370 K=1
```



```
1380  ?:"To keep a question or an answer without
      changing it,"
1390  ?"you may press <RETURN> or <ENTER>."
1400  FOR NN=1 TO N
1410  PRINT:PRINT:PRINT
1420  T$(K)=T$(NN):PRINT T$(K)
1430  GOSUB 210
1440  FOR J=1 TO 4:PRINT J;"  ";A$(J):NEXT J
1450  B(K)=B(NN):PRINT "ANSWER = ";B(K)
1460  GOSUB 300
1470  IF E=187 THEN K=K+1
1480  NEXT NN
1490  PRINT #2,"ZZZ"
1500  CLOSE #2
1510  N=K-1:GOTO 40
1520  PRINT "SAVE ANOTHER COPY OF THE QUESTIONS":G
      OTO 1250
1530  FOR K=1 TO N:PRINT K
1540  PRINT #2,T$(K);";";AB$(K);";";B(K)
1550  NEXT K
1560  PRINT #2,"ZZZ":CLOSE #2
1570  GOTO 40
1580  PRINT "PRINT A TEST":GOTO 1610
1590  PRINT "PERFORM A MULTIPLE CHOICE TEST"
1600  SC=0
1610  IF N>0 THEN 1660
1620  ?:"You need to create test questions or loa
      d a test"
1630  ?:"before you can print a test or perform the
      quiz."
1640  GOTO 70
1650  REM
1660  ?:"There are";N;"questions available."
1670  FOR J=1 TO N:S$(J)="A":NEXT J
1680  ?:"How many do you want for this test?"
1690  INPUT NT
1700  IF NT=0 THEN 40
1710  IF NT<=N THEN 1740
1720  ?:"Please enter a number from 1 to";N
1730  GOTO 1660
1740  IF EE=4 THEN E=1:GOTO 1820
1750  ?:"Do you want a hardcopy of the test"
1760  ?:"as you are performing it?"
1770  ?"  1  Print test on printer"
1780  ?"  2  Perform test on computer only"
1790  E=INP(2)
1800  IF E<49 OR E>50 THEN 1790
1810  E=E-48
1820  FOR NPROB=1 TO NT
1830  CLEARW 2:GOTOXY 0,0
```

## CHAPTER THREE

```
1840 RANDOMIZE 0
1850 X=INT(N*RND)+1
1860 IF S$(X)="B" THEN 1850
1870 PRINT T$(X):PRINT:S$(X)="B"
1880 NN=X:GOSUB 210
1890 IF E=2 THEN 1910
1900 LPRINT:LPRINT:LPRINT T$(X):LPRINT
1910 FOR J=1 TO 4:C(J)=1:NEXT J
1920 D=INT(4*RND)+1:ANS(NPROB)=D
1930 AA$(D)=A$(B(X)):C(B(X))=0
1940 FOR J=1 TO 4
1950 IF J=D THEN 1990
1960 F=INT(4*RND)+1
1970 IF C(F)=0 THEN 1960
1980 AA$(J)=A$(F):C(F)=0
1990 NEXT J
2000 FOR J=1 TO 4
2010 PRINT STR$(J);". ";AA$(J)
2020 IF E=1 THEN 2040
2030 LPRINT STR$(J);". ";AA$(J)
2040 NEXT J
2050 IF EE=4 THEN 2210
2060 PRINT:PRINT
2070 REM
2080 K=INP(2)
2090 IF K<49 OR K>52 THEN 2080
2100 K=K-48:PRINT K
2110 IF E=1 THEN LPRINT "ANSWER CHOSEN: ";K
2120 IF K=D THEN 2170
2130 PRINT "NO, THE ANSWER IS NUMBER ";STR$(D);".
"
2140 IF E=2 THEN 2190
2150 LPRINT "NO, THE ANSWER IS NUMBER";STR$(D);".
"
2160 GOTO 2190
2170 PRINT "CORRECT":SC=SC+1
2180 IF E=1 THEN LPRINT "CORRECT"
2190 PRINT:PRINT "PRESS <RETURN>."
2200 K=INP(2):IF K<>13 THEN 2200
2210 NEXT NPROB
2220 CLEARW 2:GOTOXY 0,0
2230 IF EE=4 THEN 2310
2240 PRINT "OUT OF";NT;"QUESTIONS,"
2250 "?:"YOUR SCORE IS";SC:PRINT
2260 IF E=2 THEN 70
2270 LPRINT:LPRINT
2280 LPRINT "OUT OF";NT;"QUESTIONS,"
```



```
2290 LPRINT "YOUR SCORE IS";SC
2300 GOTO 70
2310 LPRINT CHR$(12)
2320 LPRINT "ANSWER KEY":LPRINT
2330 FOR NP=1 TO NT
2340 LPRINT STR$(NP);". ";ANS(NP)
2350 NEXT NP
2360 GOTO 40
2370 END
```

# Memory Trainer

C. Regena

---

*Want to improve your memory skills? "Memory Trainer" progressively increases a sequence of numbers or letters for you to memorize and type back. See how many characters you can repeat correctly. This program works in all resolution modes—low and medium with a color RGB monitor, and high with a monochrome monitor.*

"Memory Trainer" is a painless way for adults and children alike to practice their skills of concentration and retention. When you run the program, you will have a choice of working with either letters or numbers. If you choose letters, be sure the Caps Lock key is toggled so that it produces capital letters. The computer flashes one random character on the screen. After the character disappears, type in what you have seen. If you're correct, the computer prints the same character and then adds another random character to the series. Type both of these characters in the correct order. The process continues with one new random character being added each time you get the series correct. If you miss, the computer backs up one character. The ST also keeps track of the number of characters in the longest sequence that you have entered correctly.

## **Making Choices**

You may choose the length of time the sequence appears on the screen before it's erased. Choose a number from 1 through 9, where 1 is fast and 9 is slow (lines 280–340). The number you press, E2, is then used as a delay factor in line 480.

Lines 170–270 allow you to choose either letters or numbers. The variable C is the number of possible characters, either 26 letters or ten numbers. The variable C1 is the starting ASCII code number. Line 370 uses these variables to choose the random character that will be added to the sequence.

The array A\$ keeps track of the characters in the sequence. The program is written to allow a sequence of 99 characters. If you have a good memory, you may increase this



number by changing the DIMension statement in line 30 and the limit in lines 690, 900, and 920. Most people seem to have a short-term memory limit of 5 to 9 digits for a random sequence they have seen one time. This program is a "trainer" because, each time one character is added, all previous characters stay the same. Within a few minutes you may be able to remember a sequence of 30-40 characters.

You may want to add a section of code to this program to use an entirely different sequence of random characters each time, instead of only adding a random character at the end each time.

Line 40 sets the width of the screen at 37 so that 37 characters are printed on a line. Without a width setting, the printing can go off the edge of the window. The PRINT statement will not print beyond the width setting, but INPUT can go beyond the edge of the window. In that case, you wouldn't know what you were typing. To keep your printing on the screen, INP(2) is used to receive a keyboard press. Then the character is printed. To correct your typing, you may press either the Backspace key or the Undo key. This will erase everything that you've typed so far, and you can then retype the sequence.

SOUND statements are used to play a prompting beep, to play an arpeggio for a correct answer, and to play an *uh-oh* sound for an incorrect answer. SOUND 1,0 turns off the tones.

If you enter an incorrect answer, the correct sequence will be displayed. You may then press the space bar to continue. If you prefer to stop, press the function key F10, and your score will be displayed.

### Memory Trainer

```
10 REM MEMORY TRAINER
20 DEFINT C,E,N
30 DIM A$(99)
40 FULLW 2:CLRW 2:WIDTH 37
50 GOTOXY 7,1
60 PRINT "*** MEMORY TRAINER ***"
70 PRINT:PRINT
80 PRINT "The computer will flash a series of"
90 PRINT "letters or numbers on the screen."
100 PRINT
110 PRINT "When they clear, you type in the"
120 PRINT "series in the same order, then"
130 PRINT "press RETURN or ENTER."
```

## CHAPTER THREE

```
140 PRINT
150 PRINT "Try to remember more and more!"
160 PRINT:PRINT
170 PRINT "Choose: 1 Letters"
180 PRINT TAB(10);"2 Numbers"
190 PRINT
200 E=INP(2)
210 IF E<49 OR E>50 THEN 200
220 E=E-48
230 IF E=1 THEN C=26:C1=65 ELSE C=10:C1=48
240 IF E=2 THEN PRINT "NUMBERS":GOTO 270
250 PRINT "LETTERS--Please use Caps Lock to"
260 PRINT TAB(10);"use capital letters."
270 PRINT:PRINT
280 PRINT "Choose how long to see the sequence--"
    "
290 PRINT "Press a number from 1 to 9."
300 PRINT "1 is FAST; 9 is SLOW"
310 E2=INP(2)
320 IF E2<49 OR E2>57 THEN 310
330 E2=E2-48
340 PRINT STR$(E2)
350 N=1:NN=1
360 RANDOMIZE 0
370 A$(N)=CHR$(C1+INT(C*RND))
380 CLEARW 2:WIDTH 35
390 AD$="":COLOR 5
400 GOTOXY 0,4:PRINT
410 FOR CC=1 TO N
420 FOR CD=1 TO 500:NEXT CD
430 PRINT A$(CC);
440 AD$=AD$+A$(CC)
450 NEXT CC
460 PRINT
470 COLOR 1
480 FOR CD=1 TO E2*300:NEXT CD
490 GOTOXY 0,5:PRINT SPACE$(N)
500 SOUND 1,15,1,6,5
510 SOUND 1,0
520 GOTOXY 0,7:PRINT
530 B$=""
540 FOR CD=1 TO N
550 E=INP(2)
560 IF E=8 OR E=225 THEN CLEARW 2:GOTO 500
570 IF E>96 THEN E=E-32
580 PRINT CHR$(E);
590 B$=B$+CHR$(E)
600 NEXT CD
610 IF AD$<>B$ THEN 710
620 SOUND 1,15,1,4,3
```



```
630 SOUND 1,15,5,4,3
640 SOUND 1,15,8,4,3
650 SOUND 1,15,1,5,10
660 SOUND 1,0
670 IF N>NN THEN NN=N
680 N=N+1
690 IF N>99 THEN 830
700 GOTO 370
710 GOTOXY 0,4:PRINT
720 COLOR 5
730 FOR CD=1 TO N:PRINT A$(CD);:NEXT CD
740 SOUND 1,15,5,3,10
750 SOUND 1,15,1,3,10
760 N=N-1:IF N<1 THEN N=1
770 SOUND 1,0
780 COLOR 1:WIDTH 37:GOTOXY 0,14
790 PRINT "PRESS THE SPACE BAR TO CONTINUE."
800 PRINT "PRESS F10 TO STOP."
810 E=INP(2):IF E=196 THEN 830
820 IF E=32 THEN 380 ELSE 810
830 CLEARW 2
840 GOTOXY 1,0
850 PRINT "*** MEMORY TRAINER ***"
860 PRINT:PRINT
870 PRINT "YOU ENDED WITH";N+1;"IN THE SEQUENCE
"
880 PRINT
890 PRINT "YOUR HIGHEST SCORE IS";NN
900 IF N<99 THEN 930
910 PRINT
920 PRINT "THIS PROGRAM ONLY GOES TO 99."
930 PRINT:PRINT
940 PRINT "PRESS F1 TO START OVER"
950 PRINT TAB(7);"F10 TO END"
960 E=INP(2):IF E=187 THEN 40
970 IF E<>196 THEN 960
980 PRINT:PRINT "END"
990 END
```

# Softball Statistics

Roger Felton  
Version by George Miller

---

*"Softball Statistics" makes it easy to keep track of all the individual and team results for your favorite team. You can enter data for each player's times at bat, hits, runs, and so on. The program automatically computes batting averages, stores cumulative results on disk as the season progresses, generates formatted printouts with sorted rankings for all players, and more. It runs in medium- or high-resolution mode on any Atari ST with TOS in ROM. An 80-column printer is optional but recommended.*

What's the worst position on a softball team? Catchers have to squat in an uncomfortable stance for an hour or more and duck hazardous foul balls. The pitcher has to duel with mighty sluggers and dodge powerful line drives. Whoever's playing first base has to stretch like a rubber band to nab wayward throws, while keeping at least one toe on the base. And outfielders have to scoop up bouncing grounders with the knowledge that no one is backing them up except the fence.

But as demanding as these positions are, another one may be worse—that of team statistician. Keeping track of your teammates' performance is often a laborious, thankless job. Sometimes the statistician is a reserve player or friend of the team who doesn't even get to play. Caged in the dugout, the statistician must document every hit, run, and walk, and boost team morale by contributing lively chatter. After the game, the statistician has to spend hours punching numbers into a calculator to figure out everyone's batting average.

"Softball Statistics" makes the job easier. After each game, the program prompts you to enter vital stats for each player. Then it automatically calculates the batting averages and displays sorted rankings on the screen, or it can print them out. It can also print sorted rankings for hits, runs, and runs batted in (RBIs). You can merge these game statistics with data for all previous games, and you can sort updated season



results by category and print them, too. Finally, the program lets you store the cumulative statistics on disk.

If you're a fan of professional or Little League baseball, you can use Softball Statistics to follow the fortunes of your favorite team. With modifications, it could be adapted to a wide variety of sports.

### Preparing the Program

*Be extra careful when you type Softball Statistics. A mistyped line can yield inaccurate results even if the program runs without errors.* Save a copy on disk for safekeeping before running it the first time. Softball Statistics runs in medium- or high-resolution mode on any Atari ST with the TOS operating in ROM.

Before you use the program, you'll need to enter your team's roster. Softball Statistics can handle a team with a maximum of 20 players. It stores this information in DATA statements as part of the program itself. If you're keeping stats for more than one team, keep a separate copy of the program for each team.

**Note:** Due to the way Atari BASIC stores variables, you may find it necessary to reload BASIC and the program after the latter has saved or loaded data from disk the first time.

The DATA statements for player information begin at line 2300. The statements must conform to a predefined format: a two-digit jersey number followed by a space, then the player's first or last name. Precede one-digit jersey numbers with a zero—for example, 08 for 8. Names can be any length, but only the first seven characters will appear on the printout. Each entry is separated by a comma. Here's an example:

```
2300 DATA 23 LEE,17 JACKSON,33  
      JOHNSTON,10 LONGSTREET,04  
      PICKETT
```

In the output, *JOHNSTON* and *LONGSTREET* would appear as *JOHNSTO* and *LONGSTR*.

Here, the program is listed with dummy entries in the DATA statements (such as 44 Jim and 10 PLAYERX). Substitute your own team members for these entries. If your team has fewer than 20 players, leave the remaining dummy entries in the DATA statements, but substitute the name *PLAYERX*. *The program must have 20 entries to function, and it ignores the PLAYERX entries.*

Finally, put your own team's name in the TM\$ string statement at line 190. You're ready to run.

**Important note:** You should avoid tinkering with the player-name DATA statements once you've started using the program. Otherwise, you'll have problems when the program attempts to compute cumulative season totals. If you drop a player from the roster and replace him or her with another player, the new player's totals will contain the old player's results as well. To drop a player, substitute a PLAYERX dummy entry at that position in the DATA statement. Of course, this means that the dropped player's results will no longer be included in the team totals for the season. If you want to retain a dropped player's results in the team totals, leave the player's name in the DATA statement and enter 999 in response to all input prompts for that player's stats following subsequent games (see below).

### Compiling Statistics

Once the roster is entered, you can run the program. It begins by asking for statistics for individual games. The first prompt asks

**Who did you play?**

Answer with the opposing team's name—such as Ham's Diner—and press Return. The next prompt reads

**Enter your score and their score**  
(separated by a comma):

For instance, if your team lost by a score of 9 to 5, you'd type 5,9 and press Return.

The program now begins asking for individual player statistics. If the first player name on your roster is Kevin, the program prints

**Kevin's statistics for this game:**

It then prompts you, one by one, to enter the number of times at bat, runs scored, hits, runs batted in (RBIs), doubles, triples, home runs, and walks. At each prompt, type the appropriate number and press Return. After the last prompt, the program asks

**Is everything OK (Y/N)?**



If you made any mistakes while entering the current player's stats, press N. You'll be given a chance to reenter the numbers.

When all the player's statistics are correct, press Y at the prompt. The program continues to the next player on the roster and repeats the cycle.

If a certain player missed a game, type 999 at the first prompt. This automatically enters zeros for all of that player's stats and skips to the next player. In fact, entering 999 at any prompt inputs zeros for all of a player's remaining game stats.

### Individual Printouts

After you type the last statistic for the last player, the program will print the message **WORKING** while it sorts the data. (The **WORKING** message appears at other points in the program during sorts, too, since the sort routine is written in BASIC and is not particularly fast.) In a few moments, the program says

**Do you want a printout of the game's  
stats (Y/N)?**

Type Y for yes or N for no. If you press N, the program asks if you want to input data for another game. If you press Y, it asks

**To screen or printer (S/P)?**

Type S or P. Softball Statistics then prints the individual stats for all team members for that game, sorted in descending order by batting averages (Figure 1). To pause the printout, press the left mouse button. You can resume after pausing by pressing the space bar.

The program then asks

**Do you want a sorted printout of hits,  
RBIs, and run leaders (Y/N)?**

Again, type Y for yes or N for no. If you type N, the program asks whether you want to input stats for another game. If you answer Y, it asks again whether you want the output directed to the screen or printer. It then prints sorted rankings for the various slugging categories for that game (Figure 2). As before, you can stop the output by pressing the left mouse button and restart it by pressing the space bar.

## CHAPTER THREE

**Figure 1. Printout of Team Game Stats**

ROSTER IS SORTED BY BATTING AVERAGE

#	PLAYER	AB	RUNS	HITS	RBI	2B	3B	HR	BB	AVG
09	MARTY	6	2	5	3	2	1	1	0	0.833
03	JOHN	5	2	4	2	2	0	1	1	0.800
55	MIKE	4	1	3	1	1	0	1	0	0.750
44	JIM	5	4	3	1	2	0	0	0	0.600
08	KEN	4	1	2	1	1	1	0	0	0.500
08	BOB	6	3	3	2	2	0	0	2	0.500
22	PETE	5	1	2	2	0	0	0	0	0.400
07	BILL	5	1	2	0	1	0	0	0	0.400
06	BARRY	6	2	2	0	1	0	0	3	0.333
TOTALS		46	17	26	12	12	2	3	6	0.565

**Figure 2. Printout of Slugging Stats**

HITS SORT:			RBIS SORT:			RUNS SORT:		
#	PLAYER	HITS	#	PLAYER	RBIS	#	PLAYER	RUNS
09	MARTY	5	09	MARTY	3	44	JIM	4
03	JOHN	4	03	JOHN	2	08	BOB	3
55	MIKE	3	22	PETE	2	03	JOHN	2
44	JIM	3	08	BOB	2	06	BARRY	2
08	BOB	3	44	JIM	1	09	MARTY	2
06	BARRY	2	55	MIKE	1	55	MIKE	1
08	KEN	2	08	KEN	1	08	KEN	1
22	PETE	2	07	BILL	0	22	PETE	1
07	BILL	2	06	BARRY	0	07	BILL	1
TOTAL HITS		26	TOTAL RBIS		12	TOTAL RUNS		17

Finally, the program asks

**Do you want to input stats from another game (Y/N)?**

Generally, you'll type N at this prompt unless you're entering the results of more than one game. If you type Y, the program repeats the entire process described above.



### Season Totals

Softball Statistics makes it easy for you to tabulate running totals for the entire season by storing game results on disk. After you've entered and viewed the stats for the most recent game, the program asks

**Would you like to merge in data for the year (Y/N)?**

The first time you run Softball Statistics, of course, you won't have any previous data on disk. Answer N and skip to the next prompt. During subsequent runs, answer Y to merge in data for the year. The program then requests a filename for the disk data file and merges these existing stats with the results you've entered for the latest game or games.

Season totals are then computed automatically, and the program asks

**Do you want a printout of the year's stats (Y/N)?**

If you answer Y, the program asks if you want output directed to the screen or printer, and then prints season totals for all players. This printout includes the team's wins/losses record and sorts players in descending order by batting averages (Figure 3).

**Figure 3. Printout of Season Totals**

STATISTICS FOR THE YEAR:

RECORD FOR THE YEAR: WINS:2 LOSSES:1

ROSTER IS SORTED BY BATTING AVERAGE

#	PLAYER	AB	RUNS	HITS	RBI	2B	3B	HR	BB	AVG
03	JOHN	16	10	11	11	5	4	2	3	0.688
06	BARRY	18	12	11	8	4	1	4	5	0.611
07	BILL	17	10	10	7	3	3	3	2	0.588
55	MIKE	18	10	10	10	5	3	1	4	0.556
44	JIM	18	9	9	7	5	2	1	2	0.500
08	BOB	17	12	8	7	4	1	2	1	0.471
09	MARTY	17	10	8	10	4	2	3	4	0.471
22	PETE	17	7	6	4	3	1	1	3	0.353
08	KEN	17	6	6	7	3	1	2	4	0.353
TOTALS		155	86	79	71	36	18	19	28	0.510

## CHAPTER THREE

Afterward, the program asks whether you want sorted printouts for hits, RBIs, and runs, again based on season totals (these charts resemble those in Figure 2). Finally, the program gives you the opportunity to save the updated data file on disk until the next game.

If you typed N after the previous prompt, the program asks **Do you want to save the data (Y/N)?**

If you answer Y, the program asks for a filename for the updated data file, saves the file, and then ends.

### Softball Computing

If you're interested in programming, you can learn a lot by studying Softball Statistics. It's written in straight BASIC with no machine language. In fact, the input and output routines beginning at lines 2350 and 2470 are general enough to be adapted to your own programs.

You don't have to be a programmer, though, to appreciate Softball Statistics. If you're a softball statistician, no longer do you have the worst position on the team. Maybe it's the shortstop....

### Softball Statistics

```
10  TITLE$=" Softball Statistics "+CHR$(0)
20  LP$=SPACE$(2)+"# PLAYER"+SPACE$(4)+"AB"+SPACE$(3)
30  LP$=LP$+"RUNS"+SPACE$(2)+"HITS"+SPACE$(3)+"RBI"+SPACE$(3)
40  LP$=LP$+"2B"+SPACE$(4)+"3B"+SPACE$(4)+"HR"+SPACE$(4)+"BB"+SPACE$(4)+"AVG"
50  GOSUB CLEARIT
60  IF PEEK(SYSTAB+0) <> 4 THEN 140
70  PRINT " `SOFTBALL STATISTICS` "
80  PRINT " REQUIRES A MEDIUM OR HI RESOLUTION"
90  PRINT " SCREEN.":PRINT
110 PRINT " PLEASE RESET RESOLUTION BEFORE"
120 PRINT " CONTINUING."
130 END
140 GOSUB CLEARIT:GOSUB TITLEBAR
150 D5=5
160 D6=2
170 PL=20
180 DIM B(9),CC(20),IN(21),ST(8),RT(20,8),TT(20,8),F$(8),NA$(20),R$(21)
190 TM$="Sundogs"
200 C$="0000"
```



```

210   FOR I=1 TO 8
220   READ F$(I)
230   NEXT I
240   FOR J=1 TO PL
250   READ NA$(J)
260   NA$(J)=MID$(NA$(J),1,10)
270   NEXT J
280   FOR J=1 TO PL
290   R$(J)=MID$(NA$(J),1,LEN(NA$(J)))+SPACE$(10-LEN(NA$(J)))
300   FOR I=1 TO 8
310   TT(J,I)=0
320   ST(I)=0
330   NEXT I
340   NEXT J
350   GOSUB CLEARIT:GOTOXY 5,10:PRINT "Do you want
      to:":PRINT
360   PRINT SPACE$(20);"1) Enter new statistics."
365   PRINT SPACE$(20);"2) Review disk file"
370   A = INP(2)
380   IF A = ASC("1") THEN 410
390   IF A = ASC("2") THEN 3530
400   GOTO 370
410   GOSUB CLEARIT:PRINT "GAME STATISTICS"
420   PRINT:PRINT "Who did you play"
430   INPUT OT$
440   PRINT:PRINT "Enter your score and their score
      (separated by a comma)"
450   INPUT YS,TS
460   W=W+ABS(YS>TS)
470   L=L+ABS(TS>YS)
480   FOR J=1 TO PL
490   IF MID$(NA$(J),4,7)<>"PLAYERX" THEN 520
500   R$(J)=R$(J)+"00000000000000000000000000000000
      00.000"
510   GOTO 730
520   GOSUB CLEARIT
530   PRINT MID$(NA$(J),4,LEN(NA$(J)));"s statistics
      for this game:"
540   FOR I=1 TO 8
550   B(I)=0
560   PRINT F$(I);TAB(14);
570   INPUT B(I)
580   IF LEN(STR$(B(I)))>=D5 THEN 550
590   IF B(I)<>999 THEN 640
600   FOR K=I TO 8
610   B(K)=0
620   NEXT K
630   I=8
640   NEXT I

```

## CHAPTER THREE

---

```
650 PRINT:PRINT"Is everything OK (Y/N) ?"
660 A$ = CHR$(INP(2))
670 IF A$ = "N" OR A$ = "n" THEN 520
680 GOSUB BUILD
690 FOR I=1 TO 8
700 RT(J,I)=RT(J,I)+B(I)
710 TT(J,I)=TT(J,I)+B(I)
720 NEXT I
730 NEXT J
740 GOSUB WORKING
750 MM=0
760 FOR I=1 TO 8
770 FOR J=1 TO PL
780 ST(I)=ST(I)+TT(J,I)
790 NEXT J
800 B(I)=ST(I)
810 NEXT I
820 R$(J)=" "
830 GOSUB BUILD
840 TT$=R$(J)
850 GOSUB AVERAGE:GOSUB CLEARIT
860 PRINT "Do you want to input statistics from
another game (Y/N)?"
870 GOSUB GETKEY
880 IF A$ = "Y" OR A$ = "y" THEN 280
890 GOSUB CLEARIT
900 PRINT "Would you like to merge in data for t
he year (Y/N)?"
910 GOSUB GETKEY
920 IF A$ = "N" OR A$ = "n" THEN 960
930 GOSUB CHECKFILE
940 W=SW+W
950 L=SL+L
960 GOSUB WORKING
970 FOR J=1 TO PL
980 FOR I=1 TO 8
990 IF A$="N" OR A$="n" OR MID$(NA$(J),4,7)="PLA
YERX" THEN 1040
1000 B(I)=VAL(MID$(R$(J),11+(I-1)*4,4))
1010 B(I)=RT(J,I)+B(I)
1020 RT(J,I)=B(I)
1030 GOTO 1050
1040 B(I)=RT(J,I)
1050 ST(I)=0
1060 NEXT I
1070 R$(J)=MID$(R$(J),1,10)
1080 GOSUB BUILD
1090 NEXT J
1100 MM=1
1110 FOR I=1 TO 8
```



```
1120 FOR J=1 TO PL
1130 ST(I)=ST(I)+RT(J,I)
1140 NEXT J
1150 B(I)=ST(I)
1160 NEXT I
1170 R$(J)=" "
1180 GOSUB BUILDR
1190 TT$=R$(J)
1200 GOSUB CLEARIT
1210 PRINT "Do you want a printout of the year's
      statistics (Y/N)?"
1220 GOSUB GETKEY
1230 IF A$ = "N" OR A$ = "n" THEN 1260
1240 GOSUB WORKING
1250 GOSUB AVERAGE:GOSUB CLEARIT
1260 PRINT "Do you want to SAVE the data (Y/N)?"
1270 GOSUB GETKEY
1280 IF A$ = "Y" OR A$ = "y" THEN 1300
1290 END
1300 GOTO WRITEFILE
1310 '
1320 SHELL:
1330 FOR J=1 TO PL
1340 IN(J)=J
1350 CC(J)=VAL(MID$(R$(J),BB,E))
1360 NEXT J
1370 FOR J=PL-1 TO 1 STEP -1
1380 FOR I=1 TO J
1390 IF CC(IN(I))>CC(IN(I+1)) THEN 1430
1400 TE=IN(I)
1410 IN(I)=IN(I+1)
1420 IN(I+1)=TE
1430 NEXT I
1440 NEXT J
1450 RETURN
1460 '
1470 BUILDR:
1480 IF B(1)=0 THEN 1510
1490 IF B(3)=0 THEN 1510
1500 GOTO 1540
1510 B(9)=0
1520 AV$="0.000"
1530 GOTO 1550
1540 B(9)=INT(B(3)/B(1)*1000+.5)/1000+.0001
1550 FOR I=1 TO 8
1560 B$=STR$(B(I))
1570 B$=MID$(C$,1,D5-LEN(B$))+MID$(B$,D6,LEN(B$))
1580 R$(J)=R$(J)+B$
1590 NEXT I
1600 IF B(9)=0 THEN 1660
```

## CHAPTER THREE

---

```
1610 AV$=STR$(B(9))
1620 IF MID$(AV$,1,1)<>" " THEN 1640
1630 AV$=MID$(AV$,2,6)
1640 IF MID$(AV$,1,1)<>". " THEN 1660
1650 AV$="0"+AV$
1660 R$(J)=R$(J)+MID$(AV$,1,5)
1670 RETURN
1680 '
1690 AVERAGE:
1700 BB=43
1710 E=5
1720 GOSUB SHELL
1730 IF MM=1 THEN 1770
1740 GOSUB CLEARIT
1750 PRINT "Do you want a printout of the game's
statistics (Y/N)?"
1760 GOSUB GETKEY
1770 IF A$ = "N" OR A$ = "n" THEN 1810
1780 GOSUB PRINTOPT
1790 IF DE = 1 THEN GOSUB SCREENPRNT:GOTO 1810
1800 IF DE = 2 THEN GOTO LINEPRNT
1810 RETURN
1820 '
1830 WORKING:
1840 PRINT
1850 PRINT " WORKING..."
1860 RETURN
1870 '
1880 PRINT
1890 PRINT "Do you want sorted printouts of hits,
RBI's, and run leaders (Y/N)?"
1900 GOSUB GETKEY
1910 IF A$ = "N" OR A$ = "n" THEN 1940
1920 GOSUB PRINTOPT
1930 GOTO 1950
1940 RETURN
1950 GOSUB WORKING
1960 BB=19
1970 E=4
1980 GOSUB SHELL
1990 I=3
2000 IF DE = 1 THEN GOSUB TOSCREEN ELSE GOSUB TOL
INEPTR
2010 BB=23
2020 GOSUB SHELL
2030 I=4
2040 IF DE = 1 THEN GOSUB TOSCREEN ELSE GOSUB TOL
INEPTR
2050 BB=15
2060 GOSUB SHELL
```



## Applications and Education

---

```
2070 I=2
2080 IF DE = 1 THEN GOSUB TOSCREEN ELSE GOSUB TOL
      INEPT
2090 RETURN
2100 '
2110 GETKEY:
2120 A$ = CHR$(INP(2))
2130 IF A$ = "N" OR A$ = "n" OR A$ = "Y" OR A$ =
      "y" THEN RETURN ELSE 2120
2140 RETURN
2150 '
2160 PRINTOPT:
2170 PRINT
2180 PRINT "To screen or printer (S/P)?"
2190 A$ = CHR$(INP(2))
2200 IF A$ = "S" OR A$ = "s" THEN DE = 1:GOTO 222
      0
2210 IF A$ = "P" OR A$ = "p" THEN DE = 2 ELSE 219
      0
2220 RETURN
2230 '
2240 CLEARIT:
2250 CLEARW 2:FULLW 2:GOTOXY 0,0
2260 RETURN
2270 '
2280 DATA Times at Bat,Runs,Hits,RBI's,Doubles,Tr
      iple,Home Runs,Walks
2290 REM LIST PLAYERS BY NUMBER & NAME
2300 DATA 01 Kevin,02 Tom,03 Patrick,04 Eddie,05
      Gregg
2310 DATA 06 George,07 David H.,08 David F.,09 Se
      lby,10 Mark
2320 DATA 11 Neal,12 Byron,13 Paul,14 John,15 Leo
      n
2330 DATA 16 David K,17 Mike,18 PLAYERX,19 PLAYER
      X,20 PLAYERX
2340 '
2350 REM INPUT ROUTINE
2360 CHECKFILE:
2370 ON ERROR GOTO 2600
2380 GOSUB CLEARIT
2390 PRINT"Name for data file";:INPUT FF$
2400 OPEN "I",#1,FF$
2410 INPUT #1,SW,SL
2420 FOR J=1 TO PL
2430 INPUT #1,R$(J)
2440 R$(J)=MID$(NA$(J),1,LEN(NA$(J)))+SPACE$(10-LE
      N(NA$(J)))+R$(J)
2450 NEXT J:CLOSE #1:RETURN
2460 '

```

## CHAPTER THREE

```
2470 WRITEFILE:
2480 GOSUB CLEARIT:
2490 PRINT "Name of data file to write";:INPUT FF$
2500 OPEN "O",#1,FF$
2510 PRINT#1,W
2520 PRINT#1,L
2530 FOR J = 1 TO PL
2540 PRINT #1, MID$(R$(J),11,32)
2550 NEXT J
2560 CLOSE #1
2570 END
2580 '
2590 CHECKERROR:
2600 IF ERR = 53 THEN 2620
2610 PRINT "Error Number ";ERR;" at line ";ERL:EN
D
2620 PRINT "File not found on disk drive specifie
d."
2630 CLOSE 1
2640 RESUME 2390
2650 '
2660 SCREENPRNT:
2670 GOSUB CLEARIT:PRINT:IF MM=1 THEN T$="THE YEA
R":GOTO 2690
2680 T$="THIS GAME"
2690 PRINT "STATISTICS FOR "T$":":IF MM=1 THEN GO
TO 2710
2700 PRINT TM$" VS "OT$"      Score:"YS"—"TS:GOTO 2
720
2710 PRINT "RECORD FOR THE YEAR: Wins:"W" Losses
:"L
2720 PRINT :PRINT "Roster is sorted by batting av
erage":PRINT
2730 PRINT LP$
2740 FOR J=1 TO PL:GOSUB PAUSE
2750 IF MID$(R$(IN(J)),4,7)="PLAYERX" THEN 2830
2760 PRINT SPACE$(1);MID$(R$(IN(J)),1,10);SPACE$(
1);
2770 FOR I= 1 TO 8:Q=0:FOR K=0 TO 3
2780 IF MID$(R$(IN(J)),11+(I-1)*4+K,1) <> "0" THE
N Q=1
2790 IF MID$(R$(IN(J)),11+(I-1)*4+K,1)="0" AND Q=
0 AND K=3 THEN PRINT "0";:GOTO 2820
2800 IF MID$(R$(IN(J)),11+(I-1)*4+K,1)="0" AND Q=
0 THEN PRINT " ";:GOTO 2820
2810 PRINT MID$(R$(IN(J)),11+(I-1)*4+K,1);
2820 NEXT K:PRINT SPACE$(2);:NEXT I:PRINT SPACE$(
1);MID$(R$(IN(J)),43,5)
2830 NEXT J:PRINT :PRINT " TOTALS";SPACE$(5);
2840 FOR I=1 TO 8
```



```
2850 Q=0:FOR K=1 TO 4:IF MID$(TT$(I-1)*4+K,1) <>
    "0" THEN Q=1
2860 IF MID$(TT$(I-1)*4+K,1)="0" AND Q=0 AND K=4
    THEN PRINT "0";:GOTO 2890
2870 IF MID$(TT$(I-1)*4+K,1)="0" AND Q=0 THEN PR
    INT SPACE$(1);:GOTO 2890
2880 PRINT MID$(TT$(I-1)*4+K,1);
2890 NEXT K:PRINT SPACE$(2);:NEXT I:PRINT SPACE$(
    1);MID$(TT$,33,5)
2900 PRINT :GOTO 1880
2910 '
2920 TOSCREEN:
2930 PRINT :T=0:PRINT :PRINT F$(I)" SORT:";PRINT
2940 PRINT "# PLAYER";space$(6);F$(I):FOR J=1 TO
    PL:GOSUB PAUSE
2950 IF MID$(R$(IN(J)),4,7)="PLAYERX" THEN 3020
2960 PRINT MID$(R$(IN(J)),1,10);SPACE$(4);
2970 Q=0:FOR K=0 TO 3:IF MID$(R$(IN(J)),BB+K,1) <
    > "0" THEN Q=1
2980 IF MID$(R$(IN(J)),BB+K,1)="0" AND Q=0 AND K=
    3 THEN PRINT "0":GOTO 3010
2990 IF MID$(R$(IN(J)),BB+K,1)="0" AND Q=0 THEN P
    RINT SPACE$(1);:GOTO 3010
3000 PRINT MID$(R$(IN(J)),BB+K,1);:IF K=3 THEN PR
    INT
3010 NEXT K:T=T+VAL(MID$(R$(IN(J)),BB,E))
3020 NEXT J:PRINT :PRINT "TOTAL ";F$(I);SPACE$(5)
    ;T
3030 PRINT :RETURN
3040 '
3050 LINEPRNT:
3060 LPRINT:IF MM=1THEN T$="THE YEAR":GOTO 3080
3070 T$="THIS GAME"
3080 LPRINT "STATISTICS FOR "T$":":IF MM=1 THEN G
    OTO 3100
3090 LPRINT TM$" VS "OT$" SCORE:"YS"- "TS:GOTO
    3110
3100 LPRINT "Record for the year: Wins:"W" Losse
    s:"L
3110 LPRINT :LPRINT "Roster is sorted by Batting
    Average":LPRINT
3120 LPRINT LP$
3130 FOR J=1 TO PL:GOSUB PAUSE
3140 IF MID$(R$(IN(J)),4,7)="PLAYERX" THEN 3220
3150 LPRINT SPACE$(1);MID$(R$(IN(J)),1,10);SPACE$
    (1);
3160 FOR I= 1 TO 8:Q=0:FOR K=0 TO 3
3170 IF MID$(R$(IN(J)),11+(I-1)*4+K,1) <> "0" THE
    N Q=1
```

## CHAPTER THREE

```
3180 IF MID$(R$(IN(J)),11+(I-1)*4+K,1)="0" AND Q=
    0 AND K=3 THEN LPRINT "0";:GOTO 3210
3190 IF MID$(R$(IN(J)),11+(I-1)*4+K,1)="0" AND Q=
    0 THEN LPRINT " ";:GOTO 3210
3200 LPRINT MID$(R$(IN(J)),11+(I-1)*4+K,1);
3210 NEXT K:LPRINT SPACE$(2);:NEXT I:LPRINT SPACE
    $(1);MID$(R$(IN(J)),43,5)
3220 NEXT J:LPRINT:LPRINT " TOTALS"+SPACE$(5);
3230 FOR I=1 TO 8
3240 Q=0:FOR K=1 TO 4:IF MID$(TT$(I-1)*4+K,1) <>
    "0" THEN Q=1
3250 IF MID$(TT$(I-1)*4+K,1)="0" AND Q=0 AND K=4
    THEN LPRINT "0";:GOTO 3280
3260 IF MID$(TT$(I-1)*4+K,1)="0" AND Q=0 THEN LP
    RINT SPACE$(1);:GOTO 3280
3270 LPRINT MID$(TT$(I-1)*4+K,1);
3280 NEXT K:LPRINT SPACE$(2);:NEXT I:LPRINT SPACE
    $(1);MID$(TT$,33,5)
3290 LPRINT:GOTO 1880
3300 '
3310 TOLINEPTR:
3320 LPRINT :T=0:LPRINT :LPRINT F$(I)" SORT:" :LPR
    INT
3330 LPRINT "#"+SPACE$(2)+"PLAYER"+SPACE$(6);F$(I
    ):FOR J=1 TO PL:GOSUB PAUSE
3340 IF MID$(R$(IN(J)),4,7)="PLAYERX" THEN 3410
3350 LPRINT MID$(R$(IN(J)),1,10)SPACE$(4);
3360 Q=0:FOR K=0 TO 3:IF MID$(R$(IN(J)),BB+K,1) <
    > "0" THEN Q=1
3370 IF MID$(R$(IN(J)),BB+K,1)="0" AND Q=0 AND K=
    3 THEN LPRINT "0":GOTO 3400
3380 IF MID$(R$(IN(J)),BB+K,1)="0" AND Q=0 THEN L
    PRINT SPACE$(1);:GOTO 3400
3390 LPRINT MID$(R$(IN(J)),BB+K,1);:IF K=3 THEN L
    PRINT
3400 NEXT K:T=T+VAL(MID$(R$(IN(J)),BB,E))
3410 NEXT J:LPRINT :LPRINT "TOTAL ";F$(I);SPACE$(
    5);T
3420 LPRINT:RETURN
3430 '
3440 PAUSE:
3450 IF PEEK(&HFFFC02) > 0 THEN 3450 ELSE RETURN
3460 '
3470 TITLEBAR:
3480 A# = GB : GINTIN = PEEK(A#+8)
3490 POKE GINTIN+0,PEEK(SYSTAB+8) : POKE GINTIN+2
    ,2
3500 S# = GINTIN+4 : TITLE$ = TITLE$ + CHR$(0)
3510 POKE S#,VARPTR(TITLE$) : GEMSYS(105)
3520 RETURN
```



```
3530 REVIEW:
3540 GOSUB CHECKFILE
3550 W=SW+W
3560 L=SL+L
3570 GOSUB WORKING
3580 FOR J=1 TO PL
3590 FOR I=1 TO 8
3600 IF A$="N" OR A$="n" OR MID$(NA$(J),4,7)="PLA
YERX" THEN 3620
3610 B(I)=VAL(MID$(R$(J),11+(I-1)*4,4))
3620 B(I)=RT(J,I)+B(I)
3630 RT(J,I)=B(I)
3640 GOTO 3660
3650 B(I)=RT(J,I)
3660 ST(I)=0
3670 NEXT I
3680 R$(J)=MID$(R$(J),1,10)
3690 GOSUB BUILD
3700 NEXT J
3710 MM=1
3720 FOR I=1 TO 8
3730 FOR J=1 TO PL
3740 ST(I)=ST(I)+RT(J,I)
3750 NEXT J
3760 B(I)=ST(I)
3770 NEXT I
3780 R$(J)=""
3790 GOSUB BUILD
3800 TT$=R$(J)
3810 GOSUB CLEARIT
3820 GOSUB WORKING
3830 GOSUB AVERAGE:GOSUB CLEARIT
3840 END
```

# Home Financial Calculator

Patrick Parrish  
Version by George Miller

---

*Rarely has there been a program integrating as many useful loan and investment features as "Home Financial Calculator." It's versatile, easy to use, and flexible. Rapid recalculation features make it an ideal tool for "what if" projections. A calculator mode with memory lets you solve problems not directly supported by the program, and you can pass values generated by one calculation to another. This version is for any Atari ST computer which has TOS in ROM in either medium or high resolution.*

"Home Financial Calculator" integrates a number of common financial calculations within a menu-driven package. It also features a calculator mode, or scratch pad area, where program variables can be manipulated by using common mathematical operations.

Be particularly careful when you type the long lines in this program which contain financial formulas. A mistyped program may still run, but the results it gives could be inaccurate.

When you run the program, switch to medium or high resolution. A main menu offers you a choice of Investment or Loan calculations. Type I or L to reach the appropriate sub-menu. When you're ready to exit the program, type Q to quit.

## Easy "What If" Projections

Before looking at any calculations, let's consider some basics of the program. Home Financial Calculator repeatedly uses the following parameters, or variables, in the calculation. When in the calculator mode (explained below), you'll reference these eight variables with the single letters outlined below:



T	Total (also referred to as Future Value, Total Owed, and so forth, depending on the calculation)
P	Present Value (principal)
I	Interest Rate
Y	Years
M	Months
N	Number of Periods (of either compounding, deposits, withdrawals, or payments, depending on the application)
D	Deposits
W	Withdrawals

As you work with Home Financial Calculator, the values of the eight variables are preserved until you change them. Whenever the program asks you for an input (for example, Interest), the current value of that variable is displayed (zero if no value has been entered yet). If you want to keep the current value, just press Return. Otherwise, enter the new value and press Return.

With this feature, Home Financial Calculator makes it easy for you to generate "what if" projections. Simply run the same calculation repeatedly, each time changing a previously entered value. Press Return to keep a value, and change only one or two values to see the effect on the final result.

You can also store the current value into the calculator mode's Memory Register or recall a value from the Memory Register. To see how all this works, let's take a closer look at your options.

### Your Investment Menu

When you type *I* from the main menu, the Investment submenu appears (see next page). Determine which option you want and press the appropriate key.

Each option displays screen prompts which ask you to input several values. These values are stored in the eight variables mentioned above: *T* for Total (Future Value), *P* for Present Value (principal), *I* for Interest Rate, *Y* for Years, *M* for Months, *N* for Number of Periods, *D* for Deposits, and *W* for Withdrawals. Of course, not all calculations require you to enter all these values, while others may ask for additional information.

- 1) **Future Value with Periodic Interest**
- 2) **Future Value with Interest Compounded Continuously**
- 3) **Future Value with Regular Deposits**
- 4) **Future Value with Cash Flows**
- 5) **Withdrawal of Funds**
- 6) **Net Present Value**
- 7) **Calculator Mode**
- 8) **Return to Main Menu.**

Most calculations can be solved for any *one* of the variables. To solve for a variable, enter an uppercase X at the corresponding input prompt. For example, you could enter values for everything except the Interest Rate by typing X at the Interest Rate prompt. Home Financial Calculator then solves for the Interest Rate.

Remember, however, that the program can solve for only *one* variable during each calculation. If you enter an X at more than one prompt, the program does not have enough information to calculate an answer.

### **Future Value with Periodic Interest**

Home Financial Calculator's options are fairly self-explanatory when you run the program, but let's try an example. We'll calculate the future value of an investment drawing periodic interest. This kind of investment could be a savings account or interest-bearing checking account, bonds, or a money market account. To choose this option, enter 1 at the Investment submenu.

After the screen clears, the program asks for the first input, Future Value, which appears with an asterisk (\*). Below this is a zero, the current value of this variable in memory (all variables start out with a value of zero). Following this is an input prompt.

The asterisk preceding Future Value means that this is one of the variables you can solve for. (A variable *not* preceded by an asterisk means that variable *cannot* be solved for in that particular calculation, so X would be an illegal response.) If you'd like to calculate the Future Value, enter an X



here, and answer all the other prompts with the appropriate values.

Let's calculate the future value of a \$1,000 investment drawing 8 percent interest for two years and three months, with four compounding periods each year. Enter an X for *Future Value*, since we'll be solving for this total. Answer *Present Value* with 1000 (the principal you're investing); *Annual Int Rate (%)* with 8 (enter the percentage, not a fraction); *For # Of Years* with 2; *For # Of Months* with 3; and *# Of Periods (Compounding)* with 4. After you enter the last value, Home Financial Calculator figures the *Total Future Value* and displays the answer, \$1195.09.

Now suppose you wish to know the future value of the same \$1,000 investment if you make 9 percent interest. Choose option 1 on the Investment submenu again and rerun the calculation. Notice how Home Financial Calculator automatically prints the current value of each variable at each prompt. The *Future Value* prompt shows a current value of 1195.09 from the previous calculation. Type X at this prompt, enter 9 for Interest Rate, and press Return at all other prompts to preserve their values. The result should be \$1221.71.

The versatility of Home Financial Calculator becomes apparent when you realize how many different ways you can run this calculation. Using this same menu option, you can calculate the initial investment (or present value) necessary to accrue a certain future value with periodic interest, the interest rate necessary to accrue a future value from a present value, or the time in years and months it would take to accumulate a future amount from an initial investment with periodic interest payments. Just enter an X for the unknown value you're seeking and fill in all the other prompts.

### **Future Value with Interest Compounded Continuously**

Option 2, a variation of option 1, handles investments paying a continuous interest rate. Like option 1, option 2 can handle a number of calculations. Just place an X in the slot you'd like to solve for.

Here, after entering all other parameters, you can calculate the future value of an investment, the initial investment required to reach a certain future value, the interest required to reach a desired future value, or the time required to reach a certain future value at a specified interest rate.

Notice that any variables used in option 1 will be displayed with their current values when you run option 2. Recall that the eight major variables in Home Financial Calculator retain their values throughout the program until you change them. This feature is convenient when you're going from one option to another on the Investment or Loan submenu.

In addition, the values are preserved for use in the calculator mode. For instance, you could compare the effect of continuously compounded interest to periodic interest (option 1) without having to retype the input.

### **Future Value with Regular Deposits**

If you're interested in setting up an annuity, choose option 3 on the Investment submenu. You can determine the future value of an account, such as a savings account, Individual Retirement Account, or college or vacation fund, which will receive regular deposits where interest is compounded with each deposit.

Option 3 can also tell you the amount of each deposit necessary to accrue a future value, the interest rate needed to provide some future value with regular deposits, or the time it will take to amass a future value with regular deposits.

### **Future Value with Cash Flows**

Option 4 does a single calculation. It always solves for *Future Value*, so don't enter an X anywhere. It calculates the future value of an investment with yearly cash flows (either positive or negative). The *Annual Interest Rate* you input here is the growth rate on the money you've invested.

As an example, suppose you want to determine the value of a vacation fund collected over four years. You're asked for the number of years and for the deposit or withdrawal each year. You deposit \$500 in the fund the first year and \$200 the second. The third year you are forced to withdraw \$300 (entered as -300), and the fourth year, you put in \$400. The fund has a growth rate of 12 percent. Its value after four years will be \$1,017.34.

A future value determination can also tell you whether an investment is worthwhile. If the future value of all cash flows is positive or zero, the investment is profitable. A negative future value, on the other hand, represents a losing investment.



### Withdrawal of Funds

If you intend to open an account from which you can regularly withdraw funds, choose option 5. With this option, you can determine the initial deposit required in the account to cover your withdrawals, the amount you can withdraw regularly from this account, the rate of interest you must make on funds in the account, or the period of time over which you can make withdrawals.

### Net Present Value

Option 6 lets you determine the feasibility of a prospective investment by calculating its net present value. Net present value is the current value of all future yearly cash flows to an investment along with any initial cash requirement. The interest rate you input here is the rate of return you require on your investment. A positive net present value indicates a profitable investment, while a negative result signifies a losing investment.

As an example, suppose you have the opportunity to make a \$2,000 investment which would return \$1,500 the first year, cost you \$750 the second year, and return \$1,900 the third year. You hope to make 13 percent on your money. With option 6, you would determine a net present value of \$56.87, representing a profitable investment.

### The Calculator Mode

Option 7 puts you in the calculator mode (also available from the Loan submenu). Calculator mode works much like a hand-held calculator with a single memory. You can type in a value or recall one from a variable by entering its symbol—T(otal), P(resent Value), I(nterest Rate), Y(ears), M(onths), N(umber of Periods), D(eposits), and W(ithdrawals). You can perform simple math on values stored in the Memory Register by using reverse Polish notation. And you can use the results in future calculations.

When you enter calculator mode, the calculator command line appears on the screen:

```
V S H R M+ M- M* M/ MR MC  
MEM=0
```

## CHAPTER THREE

Here are the commands:

<b>V</b>	View the values of the eight primary variables
<b>S</b>	Store Memory Register into a variable
<b>H</b>	Help—prints the command line
<b>R</b>	Return to main menu, exit calculator mode
<b>M+</b>	Add the last input to the Memory Register
<b>M-</b>	Subtract the last input from the value in the Memory Register and store the result in the Register
<b>M*</b>	Multiply the last input times the value in the Memory Register and store the result in the Register
<b>M/</b>	Divide the last input into the value in the Memory Register and store the result in the Register
<b>MR</b>	Memory Recall
<b>MC</b>	Memory Clear to zero
<b>MEM=</b>	Memory Register's current value

If you've run through a sample investment calculation, you now have some variables in memory. Enter V in the calculator mode to see them. The screen displays the eight values currently in memory for the eight variables.

To work with one of these variables, enter one of their letters (T, P, I, Y, M, N, D, or W) and press Return. Then type M+ to add it to the Memory Register (all variables must be stored in the register before you can perform any operations on them). Suppose you put the current value for T into the register and now wish to add \$229 to this value. Enter 229, press Return, type M+, and press Return. The addition is performed and the result displayed. To store this value back into the T variable, enter S for Store. A prompt appears, requesting the variable in which you intend to store the value. Type T to store the value into the variable T.

You can also use the Memory Register to hold a value not represented by any of the eight variables. To do this, determine a value using the calculator mode and store it into the Memory Register with M+. Then, when you're running a calculation elsewhere in the program, you can substitute this value for any of the eight primary variables by typing MR (Memory Recall) at the appropriate prompt. MR can be used both in the calculator mode and at any prompt where the previous value is displayed.



Finally, option 8 on the Investment submenu returns you to the main menu. Once there, you can perform some loan calculations by typing L.

### **Loan Calculations**

Here is the Loan calculations submenu:

- 1) **Regular Loan Payments**
- 2) **Remaining Loan Liability**
- 3) **Final Loan Payment**
- 4) **Single Payment Loan**
- 5) **Loan Amortization Schedule**
- 6) **Calculator Mode**
- 7) **Return to Main Menu**

### **Regular Loan Payments**

Option 1 handles a number of calculations for equal payment loans. You can figure the principal of a loan, the amount of each regular payment necessary to repay a loan, the annual interest rate on a loan with regular payments, or the term of the loan.

### **Remaining Loan Liability**

With option 2, you can determine the remaining balance on a loan with regular payments after a number of payments have been made. Enter the principal on the loan, the amount of each payment, the annual interest rate, the number of payments yearly, and the last payment number.

### **Final Loan Payment**

Option 3 calculates the amount of the final payment on a loan. In many cases, the last payment of a loan will vary from the amount of the regular payment. This option handles situations where the final payment is greater than (balloon payments) or less than the regular payment.

### **Single Payment Loan**

Option 4 calculates the amount owed on a loan that is paid off with a single payment. You must input the principal on the

loan, its annual interest rate, its term in years and months, and the number of times a year the interest on the principal is compounded.

### **Loan Amortization Schedule**

Option 5 displays a loan amortization schedule. Enter the principal on the loan, the amount of each payment, the annual interest rate, the term of the loan, and the number of payments yearly. Then enter the period of the year in which the loan began (for instance, 10 for October) and the range in years of the amortization schedule you'd like to examine.

Because of the complexity of these calculations, there may be a delay before the output appears on the screen, especially if you have chosen to look at the latter years in a long-term loan repayment schedule (such as a home mortgage). When the amortization table appears, it displays the payment number, the beginning balance for the period, the amount paid toward the loan principal, the amount paid in interest, and the ending balance. To keep the information from scrolling off the screen, the program shows only a few payment periods at a time. Press Return to view another screenful. When the end of a year is reached, the program gives the total amounts paid on the principal and interest for the year. In addition, when the last period of the loan is reached, the program displays the final payment for the loan.

The last two options on the Loan submenu are the same as those on the Investment submenu. Remember that you press Q to quit the program.

### **Modifying the Program**

Home Financial Calculator is written in a modular format for easy modification. For many routines, it uses common input labels (lines 4590–4960) and some output labels (lines 4970–5050). If you want to add an investment or loan calculation routine, choose the labels from these lines that fit your application.

Also, you may wish to add a printer option to the loan amortization schedule. Examine lines 3140–3840. Here, variable D5 (defined in line 140) determines the number of loan payments considered on each screen. Variables S1, S2, S3, and



S4 (defined in lines 150-180) format the output horizontally on the screen.

### Home Financial Calculator

```
10  GOSUB 5340
20  RES = PEEK(SYSTAB+0)
30  IF RES <> 4 THEN 60
40  ?"Please switch to Medium or High"
50  ? "Resolution.":STOP
60  COLOR 1,1
70  DIM V(8)
80  V$="TPIYMNDW"
90  C$="VSHR"
100 C0$="V S H R "
110 C1$="M+ M- M* M/ MR MC"
120 C2$="M+M-M*M/MRMC"
130 Q$=""
140 D5=12
150 S1=10
160 S2=25
170 S3=40
180 S4=55
190 TITLE$=" Home Financial Calculator "+CHR$(0)
200 GOSUB 5340:GOSUB TITLEBAR
210 PRINT "INVESTMENTS, LOANS, or QUIT? (Select
    'I','L', or 'Q') "
220 A$=CHR$(INP(2))
230 IF A$="I" OR A$ = "i" THEN 260
240 IF A$="L" OR A$ = "l" THEN 2120
245 IF A$="Q" OR A$ = "q" THEN END
250 GOTO 220
260 GOSUB 5340
270 TITLE$=" INVESTMENTS ":GOSUB TITLEBAR
280 GOTOXY 10,5:PRINT "1) FUTURE VALUE WITH PERI
    ODIC INTEREST"
290 GOTOXY 10,6:PRINT "2) FUTURE VALUE WITH INTE
    REST COMPOUNDED CONTINUOUSLY"
300 GOTOXY 10,7:PRINT "3) FUTURE VALUE WITH REGU
    LAR DEPOSITS"
310 GOTOXY 10,8:PRINT "4) FUTURE VALUE WITH CASH
    FLOWS"
320 GOTOXY 10,9:PRINT "5) WITHDRAWAL OF FUNDS"
330 GOTOXY 10,10:PRINT "6) NET PRESENT VALUE"
340 GOTOXY 10,11:PRINT "7) CALCULATOR MODE"
350 GOTOXY 10,12:PRINT "8) RETURN TO MAIN MENU"
360 GOTOXY 10,14:PRINT "YOUR CHOICE?";
370 A=INP(2)-48
380 IF A<1 OR A>8 THEN 370
390 ON A GOTO 420,680,920,1310,1500,1890,400,190
```

## CHAPTER THREE

```
400 GOSUB 4060
410 GOTO 190
420 GOSUB 5340
430 TITLE$=" FUTURE VALUE WITH PERIODIC INTEREST
   ":GOSUB TITLEBAR
440 PRINT
450 GOSUB 4590
460 GOSUB 4630
470 PRINT "*";
480 GOSUB 4720
490 PRINT "*";
500 GOSUB 4760
510 IF E=4 THEN 530
520 GOSUB 4800
530 GOSUB 4850
540 IF E<>1 THEN 570
550 V(1)=INT(V(2)*(1+V(3)/V(6))^(V(6)*Y)*100+.5)
   /100
560 GOSUB 4970
570 IF E<>2 THEN 600
580 V(2)=INT(V(1)/((1+V(3)/V(6))^(V(6)*Y))*100+.
   5)/100
590 GOSUB 5000
600 IF E<>3 THEN 630
610 V(3)=INT((V(6)*(V(1)/V(2))^(1/(V(6)*Y))-V(6)
   )*100000+.5)/100000
620 GOSUB 5030
630 IF E<>4 THEN 660
640 V(4)=LOG(V(1)/V(2))/(V(6)*LOG(1+V(3)/V(6)))
650 GOSUB 5060
660 GOSUB 5210
670 GOTO 260
680 GOSUB 5340
690 TITLE$=" FUTURE VALUE WITH INTEREST COMPOUND
   ED CONTINUOUSLY ":GOSUB TITLEBAR
700 PRINT
710 GOSUB 4590
720 GOSUB 4630
730 PRINT "*";
740 GOSUB 4720
750 PRINT "*";
760 GOSUB 4760
770 IF E=4 THEN 790
780 GOSUB 4800
790 IF E<>1 THEN 820
800 V(1)=INT(V(2)*EXP(V(3)*Y)*100+.5)/100
810 GOSUB 4970
820 IF E<>2 THEN 850
830 V(2)=INT(V(1)/EXP(V(3)*Y)*100+.5)/100
840 GOSUB 5000
```



```

850 IF E<>3 THEN 880
860 V(3)=INT(LOG(V(1)/V(2))/Y*10000+.5)/10000
870 GOSUB 5030
880 IF E<>4 THEN 660
890 V(4)=INT(LOG(V(1)/V(2))/V(3)*100+.5)/100
900 GOSUB 5060
910 GOTO 660
920 GOSUB 5340
930 TITLE$=" FUTURE VALUE WITH REGULAR DEPOSITS
":GOSUB TITLEBAR
940 PRINT
950 GOSUB 4590
960 PRINT "*REGULAR DEPOSIT $"
970 C=6
980 GOSUB 3850
990 PRINT "*";
1000 GOSUB 4720
1010 PRINT "*";
1020 GOSUB 4760
1030 IF E=4 THEN 1050
1040 GOSUB 4800
1050 GOSUB 4850
1060 IF E<>1 THEN 1090
1070 V(1)=INT(V(7)*V(6)*((1+V(3)/V(6))^(V(6)*Y)-1)
)/V(3)*100+.5)/100
1080 GOSUB 4970
1090 IF E<>3 THEN 1230
1100 V(3)=.99
1110 I=0
1120 T=INT(V(7)*((1+V(3)/V(6))^(V(6)*Y)-1)/(V(3)
/V(6)))*100+.5)/100
1130 TE=ABS(V(3)-I)/2
1140 I=V(3)
1150 IF ABS(T-V(1))/V(1)<.000005 THEN 1210
1160 IF T<V(1) THEN 1190
1170 V(3)=V(3)-TE
1180 GOTO 1120
1190 V(3)=V(3)+TE
1200 GOTO 1120
1210 V(3)=INT(V(3)*10000+.5)/10000
1220 GOSUB 5030
1230 IF E<>4 THEN 1260
1240 V(4)=LOG(V(3)*V(1)/(V(6)*V(7))+1)/(V(6)*LOG(
1+V(3)/V(6)))
1250 GOSUB 5060
1260 IF E<>7 THEN 660
1270 V(7)=INT(V(1)*(V(3)/V(6))/((1+V(3)/V(6))^(V(
6)*Y)-1)*100+.5)/100
1280 PRINT
1290 PRINT "REGULAR DEPOSITS REQUIRED:$";V(7)

```

## CHAPTER THREE

```
1300 GOTO 660
1310 GOSUB 5340
1320 TITLE$=" FUTURE VALUE WITH CASH FLOWS ":GOSU
B TITLEBAR
1330 PRINT
1340 GOSUB 4720
1350 GOSUB 4760
1360 PRINT "CASH FLOW (+/-)"
1370 PRINT
1380 V(1)=0
1390 FOR I=1 TO V(4)
1400 PRINT "CASH FLOW - YEAR #";I
1410 INPUT A$
1420 A=VAL(A$)
1430 V(1)=V(1)+A*(1+V(3))^(V(4)-I)
1440 NEXT I
1450 V(1)=INT(V(1)*100+.5)/100
1460 GOSUB 4970
1470 TE=V(1)
1480 GOSUB 5150
1490 GOTO 660
1500 GOSUB 5340
1510 TITLE$=" WITHDRAWAL OF FUNDS ":GOSUB TITLEBA
R
1520 PRINT
1530 GOSUB 4630
1540 PRINT "*REGULAR WITHDRAWAL $"
1550 C=7
1560 GOSUB 3850
1570 PRINT "*";
1580 GOSUB 4720
1590 PRINT "*";
1600 GOSUB 4760
1610 IF E=4 THEN 1630
1620 GOSUB 4800
1630 GOSUB 4850
1640 IF E<>2 THEN 1670
1650 V(2)=INT(V(8)*V(6)/V(3)*(1-(1+V(3)/V(6))^(V(6)
(6)*Y))*100+.5)/100
1660 GOSUB 5000
1670 IF E<>3 THEN 1810
1680 V(3)=.99
1690 I=0
1700 R=INT(V(2)*V(3)/V(6)*(1/((1+V(3)/V(6))^(V(6)
*Y)-1)+1)*100+.5)/100
1710 TE=ABS(V(3)-I)/2
1720 I=V(3)
1730 IF ABS(R-V(8))/V(8)<.00005 THEN 1790
1740 IF R<V(8) THEN 1770
1750 V(3)=V(3)-TE
```



```
1760 GOTO 1700
1770 V(3)=V(3)+TE
1780 GOTO 1700
1790 V(3)=INT(V(3)*10000+.5)/10000
1800 GOSUB 5030
1810 IF E<>4 THEN 1840
1820 V(4)=LOG(V(6)*V(8)/(V(6)*V(8)-V(3)*V(2)))/(V
(6)*LOG(1+V(3)/V(6)))
1830 GOSUB 5060
1840 IF E<>8 THEN 660
1850 V(8)=INT(V(2)*V(3)/V(6)*(1/((1+V(3)/V(6))^(V
(6)*Y)-1)+1)*100+.5)/100
1860 PRINT
1870 PRINT "REGULAR WITHDRAWALS: $";V(8)
1880 GOTO 660
1890 GOSUB 5340
1900 PRINT "NET PRESENT VALUE: $"
1910 PRINT
1920 PRINT "INITIAL INVESTMENT"
1930 C=1
1940 GOSUB 3850
1950 GOSUB 4720
1960 GOSUB 4760
1970 PRINT "CASH FLOW (+/-)"
1980 PRINT
1990 NV=-V(2)
2000 FOR I=1 TO V(4)
2010 PRINT "CASH FLOW - YEAR # ";I
2020 INPUT A$
2030 A=VAL(A$)
2040 NV=NV+A/((V(3)+1)^I)
2050 NEXT I
2060 NV=INT(NV*100+.5)/100
2070 PRINT
2080 PRINT "NET PRESENT VALUE: $";NV
2090 TE=NV
2100 GOSUB 5150
2110 GOTO 660
2120 GOSUB 5340
2130 TITLE$=" LOANS ":GOSUB TITLEBAR
2140 GOTOXY 21,5:PRINT "1) REGULAR LOAN PAYMENTS"
2150 GOTOXY 21,6:PRINT "2) REMAINING LOAN LIABILI
TY"
2160 GOTOXY 21,7:PRINT "3) FINAL LOAN PAYMENT"
2170 GOTOXY 21,8:PRINT "4) SINGLE PAYMENT LOAN"
2180 GOTOXY 21,9:PRINT "5) LOAN AMORTIZATION SCHE
DULE"
2190 GOTOXY 21,10:PRINT "6) CALCULATOR MODE"
2200 GOTOXY 21,11:PRINT "7) RETURN TO MAIN MENU"
2210 GOTOXY 21,13:PRINT "YOUR CHOICE?";
```

## CHAPTER THREE

```
2220 A=INP(2)-48
2230 IF A<1 OR A>7 THEN 2220
2240 ON A GOTO 2270,2690,2870,3030,3140,2250,190
2250 GOSUB 4060
2260 GOTO 190
2270 GOSUB 5340
2280 TITLE$=" REGULAR LOAN PAYMENTS ":GOSUB TITLE
BAR
2290 PRINT
2300 PRINT "*";
2310 GOSUB 4670
2320 PRINT "*";
2330 GOSUB 4890
2340 PRINT "*";
2350 GOSUB 4720
2360 PRINT "*";
2370 GOSUB 4760
2380 IF E=4 THEN 2400
2390 GOSUB 4800
2400 GOSUB 4850
2410 IF E<>2 THEN 2460
2420 V(2)=INT(V(7)*V(6)/V(3)*(1-(1+V(3)/V(6))^( -V
(6)*Y))*100+.5)/100
2430 PRINT
2440 PRINT "AMT OF PRINCIPAL:$";V(2)
2450 GOTO 2670
2460 IF E<>3 THEN 2600
2470 V(3)=.99
2480 I=0
2490 P=INT(V(7)*V(6)/V(3)*(1-((1+V(3)/V(6))^( -V(6)
)*Y))*100+.5)/100
2500 TE=ABS(V(3)-I)/2
2510 I=V(3)
2520 IF ABS(P-V(2))/V(2) < .00005 THEN 2580
2530 IF P<V(2) THEN 2560
2540 V(3)=V(3)+TE
2550 GOTO 2490
2560 V(3)=V(3)-TE
2570 GOTO 2490
2580 V(3)=INT(V(3)*10000+.5)/10000
2590 GOSUB 5030
2600 IF E<>4 THEN 2630
2610 V(4)=-LOG(1-V(3)*V(2)/(V(6)*V(7)))/(V(6)*LOG
(V(3)/V(6)+1))
2620 GOSUB 5060
2630 IF E<>7 THEN 2670
2640 V(7)=INT(V(3)*V(2)/(V(6)*(1-(V(3)/V(6)+1)^(-
V(6)*Y))*100+.5)/100
2650 PRINT
2660 PRINT "REQ PAYMENT:$";V(7)
```



## Applications and Education

---

```
2670 GOSUB 5210
2680 GOTO 2120
2690 GOSUB 5340
2700 TITLE$=" REMAINING LOAN LIABILITY ":GOSUB TI
      TLEBAR
2710 PRINT
2720 GOSUB 4670
2730 GOSUB 4890
2740 GOSUB 4720
2750 GOSUB 4850
2760 PRINT "LAST PAYMENT # WAS:"
2770 INPUT A$
2780 A=VAL(A$)
2790 FOR J=1 TO A
2800 I=INT(P*V(3)/V(6)*100+.5)/100
2810 P=P+I-V(7)
2820 NEXT J
2830 LI=INT(P*100+.5)/100
2840 PRINT
2850 PRINT "LIABILITY AFTER ";A;" PAYMENTS:$";LI
2860 GOTO 2670
2870 GOSUB 5340
2880 TITLE$=" LAST LOAN PAYMENT ":GOSUB TITLEBAR
2890 PRINT
2900 GOSUB 4670
2910 GOSUB 4890
2920 GOSUB 4720
2930 GOSUB 4930
2940 GOSUB 4850
2950 FOR J=1 TO V(6)*Y
2960 I=INT(P*V(3)/V(6)*100+.5)/100
2970 P=P+I-V(7)
2980 NEXT J
2990 LP=INT(P*100+.5)/100+V(7)
3000 PRINT
3010 PRINT "LAST PAYMENT:$";LP
3020 GOTO 2670
3030 GOSUB 5340
3040 TITLE$=" SINGLE PAYMENT LOAN ":GOSUB TITLEBA
      R
3050 PRINT
3060 GOSUB 4670
3070 GOSUB 4720
3080 GOSUB 4930
3090 GOSUB 4850
3100 V(1)=INT(V(2)*(1+V(3)/V(6))^(Y*V(6))*100+.5)
      /100
3110 PRINT
3120 PRINT "TOTAL OWED:$";V(1)
3130 GOTO 2670
```

## CHAPTER THREE

```
3140 C5=0
3150 N5=0
3160 F=0
3170 P1=0
3180 I1=0
3190 GOSUB 5340
3200 TITLE$=" LOAN AMORTIZATION SCHEDULE ":GOSUB
    TITLEBAR
3210 GOSUB 4670
3220 GOSUB 4890
3230 GOSUB 4720
3240 GOSUB 4930
3250 PRINT "# OF PAYMENTS YEARLY"
3260 GOSUB 3850
3270 PRINT "ENTER THE PERIOD OF THE YEAR IN WHICH
    THE LOAN BEGAN"
3280 INPUT N
3290 NE=N
3300 NP=(V(4)*12+V(5))/(12/V(6))
3310 NY=INT(((N-1)+NP)/V(6)+.99)
3320 PRINT "ENTER THE RANGE OF YEARS YOU'D LIKE T
    O EXAMINE (FIRST, LAST)"
3330 INPUT F1,L1
3340 IF L1<=NY THEN 3360
3350 L1=NY
3360 FOR J1=1 TO L1
3370 IF J1<F1 THEN 3390
3380 GOSUB 5250
3390 FOR J=1 TO V(6)-N+1
3400 I=INT(P*V(3)/V(6)*100+.5)/100
3410 N5=N5+1
3420 PP=V(7)-I
3430 IF J1<>NY THEN 3470
3440 IF N5<>NP THEN 3470
3450 PP=P
3460 F=1
3470 IF J1<F1 THEN 3500
3480 PRINT TAB(5);MID$(STR$(N5),2,LEN(STR$(N5))-1
    );TAB(51);INT(P*100+.5)/100;
3490 PRINT TAB(52);INT(PP*100+.5)/100;Q$;TAB(53);
3500 P=P+I-V(7)
3510 IF F=0 THEN 3540
3520 P=0
3530 J=V(6)
3540 IF J1<F1 THEN 3570
3550 PRINT I;TAB(54);INT(P*100+.5)/100;
3560 PRINT
3570 I1=I1+I
3580 P1=P1+PP
3590 C5=C5+1
```



```
3600 IF C5<>D5 THEN 3670
3610 IF J1<F1 THEN 3670
3620 GOSUB 5210
3630 GOSUB 5340
3640 C5=0
3650 IF J=V(6)-N+1 THEN 3670
3660 GOSUB 5250
3670 NEXT J
3680 IF J1<F1 THEN 3790
3690 IF F=0 THEN 3720
3700 GOTOXY 0,0
3710 PRINT "FINAL PAYMENT :$";INT((PP+I)*100+.5)/
100
3720 PRINT
3730 PRINT "TOTAL INT PAID IN YR ";J1;":$";INT(I1
*100+.5)/100
3740 PRINT "TOTAL PRINC PAID IN YR ";J1;":$";INT(
P1*100+.5)/100
3750 IF F=1 THEN 3830
3760 IF J1=L1 THEN 3830
3770 GOSUB 5210
3780 GOSUB 5340
3790 C5=0
3800 P1=0
3810 I1=0
3820 N=1
3830 NEXT J1
3840 GOTO 2670
3850 C=C+1
3860 IF C<>3 THEN 3890
3870 PRINT V(3)*100,
3880 GOTO 3900
3890 PRINT V(C),
3900 INPUT A$
3910 IF LEN(A$)<>0 THEN 3930
3920 RETURN
3930 IF A$<>"MR" THEN 3990
3940 PRINT "MEM=";M;" USE AS VARIABLE HERE (Y/N)
"
3950 INPUT A$
3960 IF A$="N" THEN 3900
3970 V(C)=M
3980 RETURN
3990 IF A$="X" THEN E=C:RETURN
4000 IF A$="x" THEN E=C:RETURN
4010 V(C)=VAL(A$)
4020 IF C<>3 THEN 4040
4030 V(C)=V(C)/100
4040 RETURN
4050 REM CALCULATOR MODE
```

## CHAPTER THREE

---

```
4060 GOSUB 5340:TITLE$=" Calculator Mode ":GOSUB
    TITLEBAR
4070 M5=0
4080 GOSUB 4410
4090 INPUT A$
4100 IF ASC(A$)>57 THEN 4130
4110 T=VAL(A$)
4120 GOTO 4090
4130 FOR I=1 TO 8
4140 IF A$<>MID$(V$,I,1) THEN 4170
4150 PRINT V(I)
4160 T=V(I)
4170 NEXT I
4180 FOR J=1 TO 6
4190 IF A$<>MID$(C2$, (J-1)*2+1,2) THEN 4210
4200 ON J GOSUB 4460,4480,4500,4520,4540,4560
4210 NEXT J
4220 FOR K=1 TO 4
4230 IF A$<>MID$(C$,K,1) THEN 4250
4240 ON K GOSUB 4290,4340,4410,4440
4250 NEXT K
4260 IF M5=0 THEN 4090
4270 M5=0
4280 RETURN
4290 FOR I=1 TO 8
4300 PRINT MID$(V$,I,1);" ";V(I)
4310 NEXT I
4320 PRINT
4330 RETURN
4340 PRINT "IN WHAT VARIABLE ";
4350 INPUT A$
4360 FOR I=1 TO 8
4370 IF A$<>MID$(V$,I,1) THEN 4390
4380 V(I)=M
4390 NEXT I
4400 RETURN
4410 COLOR 2,1:GOTOXY 0,0:PRINT C0$;" ";C1$;" MEM
    =";M:COLOR 1,1
4420 PRINT
4430 RETURN
4440 M5=1
4450 RETURN
4460 M=M+T
4470 GOTO 4570
4480 M=M-T
4490 GOTO 4570
4500 M=M*T
4510 GOTO 4570
4520 M=M/T
4530 GOTO 4570
```



```
4540 T=M
4550 GOTO 4570
4560 M=0
4570 PRINT "MEM=";M
4580 RETURN
4590 PRINT "*FUTURE VALUE $"
4600 C=0
4610 GOSUB 3850
4620 RETURN
4630 PRINT "*PRESENT VALUE $"
4640 C=1
4650 GOSUB 3850
4660 RETURN
4670 PRINT "PRINCIPAL $"
4680 C=1
4690 GOSUB 3850
4700 P=V(C)
4710 RETURN
4720 PRINT "ANNUAL INT RATE (%)"
4730 C=2
4740 GOSUB 3850
4750 RETURN
4760 PRINT "FOR # OF YEARS"
4770 C=3
4780 GOSUB 3850
4790 RETURN
4800 PRINT "FOR # OF MONTHS"
4810 C=4
4820 GOSUB 3850
4830  $Y = V(C-1) + V(C) / 12$ 
4840 RETURN
4850 PRINT "# OF PERIODS (COMPOUNDING, DEPOSITS,
WITHDRAWALS, PAYMENTS) YEARLY"
4860 C=5
4870 GOSUB 3850
4880 RETURN
4890 PRINT "PAYMENTS $"
4900 C=6
4910 GOSUB 3850
4920 RETURN
4930 PRINT "TERM OF LOAN:"
4940 GOSUB 4760
4950 GOSUB 4800
4960 RETURN
4970 PRINT
4980 PRINT "FUTURE VALUE: $";V(1)
4990 RETURN
5000 PRINT
5010 PRINT "REQUIRED INVESTMENT: $";V(2)
```

## CHAPTER THREE

```
5020 RETURN
5030 PRINT
5040 PRINT "ANNUAL INT RATE (%) REQUIRED:";V(3)*1
    00
5050 RETURN
5060 V(5)=V(4)-INT(V(4))
5070 V(5)=INT(INT(12*V(5)*10+.5)/10)
5080 V(4)=INT(V(4))
5090 IF V(5)<>12 THEN 5120
5100 V(4)=V(4)+1
5110 V(5)=0
5120 PRINT
5130 PRINT "# OF YEARS AND MONTHS:";V(4);", ";V(5)
5140 RETURN
5150 PRINT
5160 IF TE>=0 THEN 5190
5170 PRINT "THIS IS A LOSING INVESTMENT."
5180 RETURN
5190 PRINT "THIS IS A PROFITABLE INVESTMENT."
5200 RETURN
5210 PRINT
5220 COLOR 2,2:PRINT "Press any key to continue";
    :COLOR 1,1
5230 A = INP(2)
5240 RETURN
5250 GOSUB 5340
5260 PRINT "LOAN AMORTIZATION SCHEDULE FOR YR ";J
    1
5270 PRINT "PRIN $";V(2);"   RATE ";V(3)*100;"%";"
    PAYM $";V(7)
5280 PRINT
5290 COLOR 3,1
5300 PRINT TAB(5);"#";TAB(11);"BEG BAL";TAB(26);"
    PRINC";TAB(41);"INT";
5310 PRINT TAB(56);"END BAL"
5320 COLOR 1,1
5330 RETURN
5340 CLEARW 2:FULLW 2:GOTOXY 0,0
5350 RETURN
5360 TITLEBAR:
5370 A# = GB : GINTIN = PEEK(A#+8)
5380 POKE GINTIN+0,PEEK(SYSTAB+8) : POKE GINTIN+2
    ,2
5390 S# = GINTIN+4 : TITLE$ = TITLE$ + CHR$(0)
5400 POKE S#,VARPTR(TITLE$) : GEMSYS(105)
5410 RETURN
```



## CHAPTER FOUR

# BASIC Programming





# ST Hints and Tips

George Miller

---

*With these four tricks, you can set up autobooting programs, customize your GEM desktop, read a joystick from ST BASIC, and soup up BASIC's performance with machine language subroutines. All the techniques work on the 520ST and 1040ST.*

The Atari ST computers are extremely powerful and complex machines. The numerous demo programs that are widely available offer only peeks at their true capabilities. For programmers, however, the ST's power can be frustrating because it's so elusive. Virtually no technical documentation is supplied with the ST, and the two languages it comes with—Logo and ST BASIC—have their shortcomings.

If you invest \$300 for an Atari development system package, you receive an assembler, a C compiler, and a huge mass of documentation on the Graphics Environment Manager (GEM), but most of it is not even ST-specific. The material refers to GEM as implemented on the IBM PC.

However, careful study of this mountain of paper can reveal quite a few "secrets" about the ST. Here are a few of these tricks which will enhance the power of your computer.

## **Autobooting Programs**

Have you ever wished that a certain program—perhaps a RAM disk utility, an application, or a language—could run automatically when you start up your ST? This feature would be especially handy if you needed to set up a disk for someone who wants to run a program without understanding anything more than how to turn on the computer.

The eight-bit Atari computers can automatically load and run programs by using AUTORUN.SYS files. Apple has the HELLO program, PC-DOS and AmigaDOS have batch files, and the Commodore 128 has provision for autobooting. Although it's not documented, so does your ST. You can find clues on creating an auto-execute file in GEMDOS.

As part of the initialization sequence, the ST looks for a folder called AUTO on the boot disk. Any files with a .PRG

extender found in the AUTO folder are executed in sequence. These files are known as COMMAND.PRG files.

It's easy to set up an autoboot program. Place your boot disk in your drive, then point to the File heading on the menu bar. Select the New Folder option and create a folder named AUTO.

Move any program you want to autoboot into this folder. Anytime you boot your ST from this disk, the program you have placed in the AUTO folder will run automatically. This technique works with TOS in ROM or with the earlier disk-loaded TOS. It's the most foolproof autorun system yet.

There are two caveats: (1) A program that has GEM features (drop-down menus, windows, and so on) cannot be autostarting since GEM itself has not been activated at this stage in the boot process; (2) The AUTO folder is not completely dependable when the current version of the operating system is used. Sometimes a program in an AUTO folder won't execute, especially if there are several programs in the folder. Perhaps this will be corrected in a future revision of TOS.

### Customizing the Desktop

Have you ever tried renaming your disk icons using the Install Drive option from the Options menu? Some characters can't be used, and you can't do anything with the trash can.

There is a way to change the names to anything you want. After saving your desktop, you can edit the file which stores the information for these options—DESKTOP.INF. For now, we'll use this technique to change just the icon names. Be careful not to change any other characters in the file.

First, you'll need a text editor, such as *Mince* or *EMACS*, or even a word processor like *ST Writer*. If you're using a word processor, set the left and top margins to zero.

The job itself is rather easy. Load the file DESKTOP.INF. It should look something like Figure 1.

Each character in this file gives information about your desktop. Any change will affect what you see on the desktop and, to a certain extent, even how your ST functions. Use caution, since some changes might not yield the results you expect. To be safe, make sure you're working with a backup copy of your boot disk. Store the original in a safe location. (This is always a good idea when you're experimenting with



Figure 1. DESKTOP.INF

```
#a000000
#b001100
#c77700070007000700552005055522207
 70557075057705504112306
#d
#E 9B 03
#W 00 00 0C 01 1D 16 08 A: \ *.* @
#W 00 00 28 01 1F 17 00 @
#W 00 00 0E 09 2A 0B 00 @
#W 00 00 0F 0A 2A 0B 00 @
#M 00 02 00 FF A FLOPPY DISK@ @
#M 00 03 00 FF B FLOPPY DISK@ @
#T 00 07 02 FF TRASH CAN@ @
#F FF 04 @ *.* @
#D FF 01 @ *.* @
#G 03 FF *.PRG@ @
#F 03 04 *.TOS@ @
#P 03 04 *.TTP@ @
```

any file on a disk, especially when you're modifying files that control the operation of your ST.)

Now, move the cursor to the first line that begins with #M. Change the text, replacing the words *FLOPPY DISK*, so the line reads like this:

```
#M 00 02 00 FF A Disk A@ @
```

Then change the next line to

```
#M 00 03 00 FF B Disk B@ @
```

If you want, you may change the name of the trash can icon. I called mine *Black Hole!* as a constant reminder that, unlike the Amiga or Macintosh, the ST trash can does not let you easily recover files which are deleted. (There are some disk utilities available that allow you to recover trashed files under limited conditions.)

To change the trash can icon, modify the next line to read

```
#T 00 07 02 FF Black Hole!@ @
```

The revised DESKTOP.INF file should be similar to Figure 2.

Finally, save the file back to the disk as DESKTOP.INF. The file must be saved in ASCII format, so make sure your text editor or word processor has this feature. If you're using

Figure 2. Revised DESKTOP.INF

```
#a000000
#b001100
#c77700070007000700552005055522207
70557075057705504112306
#d
#E 9B 03
#W 00 00 0C 01 1D 16 08 A: \ *.* @
#W 00 00 28 01 1F 17 00 @
#W 00 00 0E 09 2A 0B 00 @
#W 00 00 0F 0A 2A 0B 00 @
#M 00 02 00 FF A Disk A @ @
#M 00 03 00 FF B Disk B @ @
#T 00 07 02 FF Black Hole! @ @
#F FF 04 @ *.* @
#D FF 01 @ *.* @
#G 03 FF *.PRG @ @
#F 03 04 *.TOS @ @
#P 03 04 *.TTP @ @
```

*ST Writer* or some other word processors, it may be necessary to print the file to the disk in order to save it in ASCII format. With *1ST Word*, turn off WP mode, and be sure the top and left margins are set to zero.

### Reading the Joystick

ST BASIC is a fairly generic BASIC that has very few ST-specific commands. One of the most noticeably missing commands when you're trying to write a game is a function for reading the joystick. The ST works with any of the joysticks sold for eight-bit Atari and Commodore computers, but there's no STICK or STRIG function as in eight-bit Atari BASIC.

Actually, a joystick command does exist in the ST, but it's hidden deep within GEMDOS in the BIOS (Basic Input/Output System). This is an area not readily available from ST BASIC unless you use a few special techniques.

An easy way to find out what the joystick is doing is to ask the intelligent keyboard device (IKBD). The keyboard has its own microprocessor—a 6301 chip, which is a member of the 6800 family. The keyboard processor is really a small computer system, with input/output lines, RAM, ROM, and even a serial interface that handles traffic to and from the 68000



central processing unit. The 68000 is not responsible for continuously polling the keyboard for activity. The 6301 notifies the 68000 via an interrupt when anything needs processing. In addition to reading the keyboard, the 6301 reads the mouse and the joystick, and performs other functions.

The ST's link to the keyboard processor is through a chip called an ACIA (Asynchronous Communications Interface Adapter). The control register for the keyboard ACIA is located at memory address \$FFFC00. The data register is at location \$FFFC02. If you've moved to the ST from an earlier eight-bit computer, these may be the biggest hexadecimal numbers you've ever seen. Remember that the 68000 microprocessor in the ST has 24 address lines, enough for over 16,000,000 bytes of memory, as compared to the 65,536-byte maximum for earlier computers with only 16 address lines. (However, due to design limitations in the MMU (Memory Management Unit) of the ST, a maximum of 4,000,000 bytes of memory may be addressed.) For the ST, you must become accustomed to seeing hexadecimal addresses that are six digits long.

The following program is a short ST BASIC routine that reads the values of the joystick plugged into port 1 (the rear joystick connector).

```
70 POKE &hfff02,&h0012 'turn off
   mouse
80 POKE &hfff02,&h0014 : joystick =
   PEEK(&hfff02)
90 IF joystick = 511 THEN ? "north"
100 IF joystick = 2559 THEN ?
    "northeast"
110 IF joystick = 2303 THEN ? "east"
120 IF joystick = 2815 THEN ?
    "southeast"
130 IF joystick = 767 THEN ? "south"
140 IF joystick = 1791 THEN ?
    "southwest"
150 IF joystick = 1279 THEN ? "west"
160 IF joystick = 1535 THEN ?
    "northwest"
170 IF joystick < 0 THEN ? "fire button"
180 POKE &hfff02,&h0008 'turn on
   mouse
190 GOTO 70
```

Line 70 sends a command to the IKBD, via the data register at \$FFFC02, instructing it to turn off the mouse. (Note that ST BASIC uses &h to indicate hexadecimal numbers.)

Line 80 sends a command via the same address to turn on the joystick. Every movement of the joystick is reported to the processor. The joystick position is read by PEEKing the value returned in \$FFFC02.

Lines 90–170 interpret the values returned from the IKBD.

Line 180 turns the mouse back on again. This should be done before exiting the program so that the user will have control of the mouse when returning to BASIC or the desktop.

Line 190 makes the routine an infinite loop. You'll need to press Control-C to stop this demonstration. If the mouse pointer isn't visible on the screen when the program stops, enter the following line and press Return to make the pointer reappear:

**POKE &hffff02,&h0008**

To adapt this routine for use in your own programs, replace line 190 with 190 RETURN. Then use GOSUB 70 to call the routine. Replace the PRINT statements in lines 90–170 with statements to perform the desired actions when the joystick is pressed in the indicated direction.

### Mixing BASIC and Machine Language

To add real speed and power to any BASIC, it's often necessary to use machine language (ML) routines for certain tasks. In ST BASIC, you can run machine language routines by using the CALL statement. The syntax for CALL is

**CALL *address variable,parameter list***

The *address variable* is a variable that holds the memory address of the beginning of the machine language routine. This location may be the address where the routine was loaded by using the BLOAD command, or it may be the address where the ML routine was POKEd. The *parameter list* is a list of values that can be passed to the ML routine. Some routines don't require any values to be passed, so this is optional.

The program below demonstrates how to POKE an ML routine into a variable, then use the VARPTR function to find the address to CALL.



As your library of ML routines expands, you'll find this method useful. Although the example program does nothing but print the letter *A* on the left side of the menu bar, it does demonstrate that ML routines give you full access to the ST, since the menu bar is usually off-limits to BASIC.

```
110 CLEARW 2 : FULLW 2
120 GOSUB init
130 ' ML opcodes in DATA statements
140 DATA &h3f3c,&h0041,&h3f3c,
    &h0002,&h4e41,&h588f
150 DATA &h3f3c,&h000d,&h3f3c,
    &h0002,&h4e41,&h588f
160 DATA &h3f3c,&h000a,&h3f3c,
    &h0002,&h4e41,&h588f,&h4e75
170 FOR i = 1 TO 19 : READ a : POKE
    x+(i*2),a : NEXT : 'POKE ml into
    ml$
180 CALL x
190 END
200 init : ml$="This is a dummy
    variable."
210 x = VARPTR (ml$)
220 RETURN
```

These tricks demonstrate only a small part of the ST's potential. Carefully studying the documentation reveals that some extremely powerful programming techniques are lurking just below the surface. If you're a curious programmer, explore GEM for ways to use the ST's features from within the tight BASIC framework.

# ST BASIC Sorting Algorithms

C. Regena

---

*Each of these seven sorting routines is written as a subroutine that you'll find handy to add to your own programs.*

The function of a computer is to process information. Often, you'll find that you need to arrange, or sort, data. For example, you may want to interpret your raw data by sorting a list of people by age, or you may want to alphabetize a list of names. There are nearly as many sort routines as there are programmers, and the names of the sort algorithms vary as well. For the ST BASIC sort algorithms listed here, I've used representative names that correspond to how they operate.

Each sort routine is written as a subroutine that may be called by the main program. The numbers to be sorted are in the A array, and there are N number of elements. The numbers will be sorted in ascending order (1, 2, 3, ...). If you prefer to sort in descending order, you may simply print the results in reverse order. To sort strings instead of numeric values (or to alphabetize), use dollar signs for all variables used to hold strings. You may also define variable names as strings with a DEFSTR statement near the beginning of the program. See the section on sorting strings for more information.

SWAP is a handy command in ST BASIC that is often used in sort routines. SWAP *a,b* puts the value of *a* into *b*, and the value that was in *b* will become *a*. Two values are interchanged without your having to use a dummy holding variable.

## Demonstration Program

To illustrate how these sort routines are used, you may use the following main "Demo" program. Forty numbers are chosen randomly and printed on the screen. These numbers are in the A( ) array, and N=40. Line 190 calls the sorting subroutine; then the sorted numbers are printed.



### Program 1. Demo

```
100 REM DEMO FOR SORTS
110 DIM A(40)
120 FULLW 2: CLEARW 2: WIDTH 37
130 N=40
140 RANDOMIZE 0
150 FOR C=1 TO N
160 A(C)=INT(100*RND):PRINT A(C);
170 NEXT C:PRINT:PRINT
180 REM
190 GOSUB SORT1
200 REM
210 FOR C=1 TO N:PRINT A(C);:NEXT C
220 FOR C=1 TO 7:PRINT:NEXT C
230 END
```

You may want to use this program to compare the sort routines. First, type in the main program and enter SAVE DEMO. For the first sort routine, "Bubble Sort," simply add the SORT1 subroutine to the main program and then type RUN. For any of the later sort routines, type the subroutine (and save it if you wish). Then type MERGE DEMO to combine the main program with the particular sort subroutine. Edit line 190 to call the correct subroutine name and type RUN.

### Bubble Sort

The *bubble sort* is often called an interchange sort. It passes through the numbers comparing pairs. If one number is larger than the next, the two numbers are interchanged. If a switch has been made during a pass through all the numbers, the loop of comparisons starts over. The number of passes depends on how many items are out of order. The smaller numbers "bubble" toward the top. Program 2 is a version of the bubble sort that keeps track of where a switch is made so that the entire list doesn't need to be compared with each pass.

This sort is common because it is easy to understand and to program. It can be fairly quick for short lists or for lists of numbers that are not very much out of order. However, it can be slow for long lists of very mixed-up numbers.

### Program 2. Bubble Sort

```
1000 SORT1:REM BUBBLE
1010 L=N-1
1020 S=0
1030 FOR C=1 TO L
1040 IF A(C)<=A(C+1) THEN 1060
1050 SWAP A(C),A(C+1):S=1:L=C
1060 NEXT C
1070 IF S=1 THEN 1020
1080 RETURN
```

### Insertion Sort

The *insertion sort* looks at each element of a list in turn and inserts the element into its proper order in the preceding list, moving elements to make room for the inserted number. For example, the computer looks at the second element and compares it with the first element. Then it places the number either first or second, depending on its value. The third element then needs to be inserted into the ordered list of two numbers. At this point, you have three elements in order at the beginning of the main list of numbers. Next, the fourth element is inserted into the ordered list of three numbers. This process continues through the list for each element.

### Program 3. Insertion Sort

```
2000 SORT2:REM INSERTION
2010 FOR C=1 TO N-1
2020 D=A(C+1)
2030 FOR K=C TO 1 STEP -1
2040 IF D>=A(K) THEN 2070
2050 A(K+1)=A(K)
2060 NEXT K:K=0
2070 A(K+1)=D
2080 NEXT C
2090 RETURN
```

### Selection Sort

This third sort algorithm could be called a search routine or *selection* method. Originally, this sort used another array for the sorted list. The first pass through the list of numbers would find the minimum number and place it in the first element of the new array. The second pass would select the second smallest element and put it in the second place of the



new array. The process would then continue for all the elements. A certain number of passes through the original list would be required, no matter how much out of order the list might be.

Program 4 improves this minimum search method to save memory by using only the original array rather than a new array. During each pass through the numbers, both the minimum and the maximum numbers are selected and are placed at the appropriate ends. The other numbers are shifted inward. The sorted list is filled in from the ends toward the center.

### Program 4. Selection Sort

```
3000  SORT3:REM SELECTION
3010  M=N:S=1
3020  L=A(S):J=S:U=L:K=S
3030  FOR T=S TO M
3040  IF A(T)>U THEN U=A(T):K=T
3050  IF A(T)<L THEN L=A(T):J=T
3060  NEXT T
3070  IF J=M THEN J=K
3080  SWAP A(M),A(K):M=M-1
3090  SWAP A(S),A(J):S=S+1
3100  IF M>S THEN 3020
3110  RETURN
```

### Heap Sort

The *heap sort* builds a binary tree structure of numbers where each number is greater than the numbers under it. If you draw the structure, it looks like a heap. To get the final list, you need fewer comparisons than with either the bubble sort or insertion sort because you know "parent" numbers are larger than "offspring" numbers.

### Program 5. Heap Sort

```
4000  SORT4:REM HEAP
4010  C=N
4020  FOR L=INT(N/2) TO 1 STEP -1
4030  D=A(L)
4040  GOSUB CHECK
4050  NEXT L
4060  L=1
4070  FOR C=N-1 TO 1 STEP -1
4080  D=A(C+1)
4090  A(C+1)=A(1)
```

```
4100 GOSUB CHECK
4110 NEXT C
4120 RETURN
4130 CHECK: REM COMPARISON SUBROUTINE
4140 J=L
4150 K=J+J
4160 IF K>C THEN 4230
4170 IF K=C THEN 4190
4180 IF A(K+1)>A(K) THEN K=K+1
4190 IF D>=A(K) THEN 4230
4200 A(J)=A(K)
4210 J=K
4220 GOTO 4150
4230 A(J)=D
4240 RETURN
```

### Shell Sort

The *Shell sort* is a popular sort routine named for its developer, Donald Shell. Again, fewer comparisons are necessary because you know some of the numbers are ordered, and you keep track of where interchanges are made to further reduce comparisons.

#### Program 6. Shell Sort

```
5000 SORT5:REM SHELL
5010 B=1
5020 B=2*B:IF B<=N THEN 5020
5030 B=INT(B/2):IF B=0 THEN 5080
5040 FOR M=1 TO N-B:C=M
5050 D=C+B:IF A(C)<=A(D) THEN 5070
5060 SWAP A(C),A(D):C=C-B:IF C>0 THEN 5050
5070 NEXT M:GOTO 5030
5080 RETURN
```

### Determining Which Sort to Use

If you know what kind of list you are starting with, you can select which kind of sort might be the most efficient. In general, the selection sort is faster than the bubble sort. Only a certain number of passes through the numbers are required, no matter how much out of order the numbers are. If only a few numbers are out of order, the bubble sort could be quicker (even for a long list) because fewer passes would be required.



### Quick Sort

The *quick sort* has become popular in the last few years because it is one of the fastest sorting procedures in BASIC for a general list (one in which you do not know how the original list is ordered). Program 7 has been adapted for ST BASIC.

#### Program 7. Quick Sort

```
6000 SORT6:REM QUICK
6010 S(1)=1:S(2)=N:T=1
6020 IF T=0 THEN 6150
6030 T=T-1:C=2*T:L=S(C+1)
6040 M=S(C+2):X=A(L):J=L:K=M+1
6050 K=K-1:IF K=J THEN 6110
6060 IF X<=A(K) THEN 6050
6070 A(J)=A(K)
6080 J=J+1:IF K=J THEN 6110
6090 IF X>=A(J) THEN 6080
6100 A(K)=A(J):GOTO 6050
6110 A(J)=X:IF M-J<2 THEN 6130
6120 C=2*T:S(C+1)=J+1:S(C+2)=M:T=T+1
6130 IF K-L<2 THEN 6020
6140 C=2*T:S(C+1)=L:S(C+2)=K-1:T=T+1:GOTO 6020
6150 RETURN
```

### Sorting Strings

All of these sorts can be converted for use with strings instead of numbers. You may define A and all other variables that are set equal to A to be strings by using DEFSTR at the beginning of the program, such as

```
102 DEFSTR A,X
```

You may also use the dollar sign to designate a string variable name. The sort routines will then alphabetize words or names.

This last program illustrates the use of strings with the quick sort. Forty random letters are chosen and printed in the example. In your own programs, you may be using names of people or other words which you want alphabetized.

#### Program 8. String Sort

```
100 REM DEMO FOR SORT WITH STRINGS
110 DIM A$(40)
120 FULLW 2:CLEARW 2:WIDTH 36
130 N=40
140 RANDOMIZE 0
150 FOR C=1 TO N
```

## CHAPTER FOUR

---

```
160 A$(C)=CHR$(26*RND+65):PRINT A$(C); " ";
170 NEXT C:PRINT:PRINT
180 REM
190 GOSUB SORT6S
200 REM
210 FOR C=1 TO N:PRINT A$(C); " ";:NEXT C
220 FOR C=1 TO 8:PRINT:NEXT C
230 END
6000 SORT6S:REM QUICK SORT WITH STRINGS
6010 S(1)=1:S(2)=N:T=1
6020 IF T=0 THEN 6150
6030 T=T-1:C=2*T:L=S(C+1)
6040 M=S(C+2):X$=A$(L):J=L:K=M+1
6050 K=K-1:IF K=J THEN 6110
6060 IF X$<=A$(K) THEN 6050
6070 A$(J)=A$(K)
6080 J=J+1:IF K=J THEN 6110
6090 IF X$>=A$(J) THEN 6080
6100 A$(K)=A$(J):GOTO 6050
6110 A$(J)=X$:IF M-J<2 THEN 6130
6120 C=2*T:S(C+1)=J+1:S(C+2)=M:T=T+1
6130 IF K-L<2 THEN 6020
6140 C=2*T:S(C+1)=L:S(C+2)=K-1:T=T+1:GOTO 6020
6150 RETURN
```



# Custom Title Bars for ST BASIC

George Miller

---

*Use this short program to put a custom title on ST BASIC's Output window. It works on all Atari ST-series computers.*

ST BASIC puts four windows on the screen—Command, List, Edit, and Output. The Output window is where your programs actually run. This window always displays the same title at the top of the screen—Output. By now, you may be tired of staring at this title bar and probably wish there was a way to change it.

Fortunately, there is a way. There's no built-in BASIC command, however. You'll have to call a routine in a part of the ST's operating system known as AES (Application Environment Services). The job isn't difficult, but the ST BASIC manual lacks the necessary information for making system calls.

When you're programming the ST, it's helpful to remember that the operating system contains many routines which can be of help. These routines are part of GEM, the Graphics Environment Manager, which is divided into two sections: AES and VDI (Virtual Device Interface). These libraries contain almost all the routines necessary to handle screen output. Although VDI and AES routines are most easily accessed by programmers using C or machine language, ST BASIC programmers can also call them with the VDISYS and GEMSYS commands. It requires a little extra effort, though.

The short routine listed here, "Custom Title Bars," is an example of a GEMSYS call to the AES library. You can insert it into any ST BASIC program to display your program's title on the Output window's title bar.

## Modifications

Run the routine to see what it does; then modify it in the following ways when you use it in your own programs:

1. Change line 20 to assign to the string variable *title\$* the name you want to have displayed in the title bar.
2. Delete line 40, the END statement, and insert your own program at this point. However, be sure you insert an END statement at the end of your program and before line 63000. Otherwise, your program will fall through into the subroutine and cause an error.

Before actually making the GEMSYS call in line 64040, the routine POKES several parameters into system variables at the addresses pointed to by the built-in BASIC variable *gintin*. The parameters are required by this AES routine. The setup is done in lines 64010–64040.

You can find more information about calling VDI and AES routines in the Atari documentation available to software developers and in *COMPUTE!'s ST Programmer's Guide* (COMPUTE! Books, 1986).

### Custom Title Bars

```
10    FULLW 2 : CLEARW 2
20    title$="new title"
30    GOSUB titlebar
40    END
64000 titlebar : ' New title for OUTPUT window routine
64010 a# = gb : gintin = PEEK(a#+8)
64020 POKE gintin+0,PEEK(systab+8) : POKE gintin+2
      ,2
64030 s# = gintin+4 : title$ = title$ + chr$(0)
64040 POKE s#,varptr(title$) : GEMSYS(105)
64050 RETURN
```



# Adding System Power to ST BASIC

Kevin Mykytyn

---

*ST BASIC lacks commands for certain operations, such as reading the mouse pointer, but you can fill in these gaps by calling system routines with the VDISYS command. After an explanation of VDI routines and a demonstration of a useful graphics routine, you'll learn how to read the mouse pointer with VDISYS and find a program for creating your own custom mouse pointers.*

You've probably heard at least two of the three-letter acronyms associated with the Atari ST computer: TOS stands for Tramiel Operating System, a huge system program which, at the most fundamental level, allows the computer to function. And GEM stands for Graphics Environment Manager, a separate system program that handles the ST's graphics-oriented desktop. GEM, in turn, consists of three separate parts: the VDI (Virtual Device Interface), a low-level graphics interface that also handles mouse input; the AES (Application Environment Services), which uses the VDI to manage data and the desktop; and GEMDOS, which handles disk operations.

Interesting, you may say, but what's the point? For most BASIC programming, you needn't worry about TOS, GEM, VDI, AES, or GEMDOS, any more than the average driver needs to know exactly how an automobile engine works. These system programs are the invisible machinery that makes everything else happen.

However, as you may have discovered, ST BASIC lacks commands to do certain tasks such as drawing a circle or sensing the position of the mouse pointer. That's what makes one of these strange-sounding names, the VDI, an invaluable asset for BASIC programmers. The VDI holds a treasure trove of system routines that can do everything from drawing boxes and circles to rotating character fonts and manipulating raster blocks. With ST BASIC's VDISYS command, you can access

all of these routines. This compensates in large part for the missing ST BASIC commands.

### VDISYS to the Rescue

In simplest terms, the VDISYS command calls (activates) a VDI system routine to do a task that would be difficult or impossible to perform in BASIC. Furthermore, these system routines execute very quickly—a real plus when you're working with graphics. Whether executed in immediate or program mode, the VDISYS command always takes this general form:

#### VDISYS(*x*)

In this example a simple variable named *x* appears in the parentheses. It doesn't matter what value this variable represents; it's a *dummy* parameter, needed only to satisfy the syntax of the command. Don't try to enter this command yet. If you do, there's a good chance you'll see the bomb symbol that signals a system crash. A certain amount of preparation is always needed before you execute VDISYS.

When a VDISYS command is executed, control passes from your BASIC program to an internal VDI handler, which eventually passes control to the VDI routine itself. But first, the VDI handler looks at certain sections of the computer's memory, called *parameter blocks*. The data in the parameter blocks tells the handler which particular VDI routine you want to execute. There's also other information that the VDI routine itself will need. If you don't supply all the information needed to call a routine, the VDI handler can't carry out your request.

### VDI Opcodes

The first thing you must tell the computer is which VDI routine you want to call. Each VDI routine is identified by a unique *opcode number*. For instance, the VDI routine used in Program 1, "Bar Drawing," has the opcode 11. This is a generalized shape-drawing routine. (There are hundreds of VDI opcodes and associated parameters, so we don't have room here to list them. But you can find a 42-page list of selected VDI opcodes in *COMPUTE!'s ST Programmer's Guide*, available from COMPUTE! Publications.)



### Program 1. Bar Drawing

```
10 fullw 2:clearw 2:color 2,2,2
20 poke contrl,11 'VDI opcode
30 poke contrl+2,2 'number of vertices
35 poke contrl+6,0 'number of attributes
40 poke contrl+10,1 'primitive ID of bar command
50 poke ptsin,50 'x coordinate of top left corner
60 poke ptsin+2,50 'y coordinate of top left corner
70 poke ptsin+4,100 'x coordinate of bottom right corner
80 poke ptsin+6,100 'y coordinate of bottom right corner
90 vdisys (0)
```

Once you know a VDI routine's opcode number, that value must be POKEd into a special place in memory defined by the reserved variable CONTRL. Try typing PRINT CONTRL in immediate mode. Even if you haven't given this variable any value, the computer prints a number on the screen. ST BASIC always predefines CONTRL along with several similar variables. The CONTRL variable represents an actual location in memory.

Since the system automatically substitutes this location for the keyword CONTRL, you don't have to memorize a series of numbers or worry about where this parameter block really resides. To select VDI routine 11, for instance, you simply execute POKE CONTRL,11.

### How Many Corners?

Once you've POKEd the VDI opcode 11 into CONTRL, you must tell the computer how many vertices (corners) are needed to define the graphic shape you want to draw. Regular geometric shapes require different numbers of vertices. A triangle, for instance, requires a minimum of three corners. A rectangle, on the other hand, can be defined with only two—the upper left corner and the lower right one. Of course, a rectangle has a total of four corners, but the total is not what we're looking for. The computer cares only about the *minimum* number of vertices it takes to draw the shape in question. After you determine how many vertices are needed, that value



is POKEd into the location defined by `CONTRL+2`. For example, in line 30 of Program 1, the statement `POKE CONTRL+2,2` tells the computer that you want to draw a rectangle (defined by only two corners).

Notice that the second POKE is directed two bytes higher in memory than the first. Now you can see the parameter block begin to take shape: It's simply a segment of memory where you place a collection of values. The first byte of the parameter block is defined by `CONTRL`, and the remaining locations are defined as even-numbered offsets above that starting spot (`CONTRL+2`, `CONTRL+4`, and so forth).

The particular routine used in this program (termed a *generalized drawing primitive*) contains several subroutines (also called *subfunctions*), each of which performs a different drawing task. To choose a subroutine, you must POKE its identifying number (called the *primitive ID*) into the location defined by `CONTRL+10`. In this case, we want to use the bar-drawing subroutine, whose primitive ID happens to be 1. So, in line 40 of the program, we POKE `CONTRL+10,1`.

### PTSIN and INTIN

The next step is to tell the VDI handler where to place the graphic shape. Remember that you told the computer earlier how many vertices it takes to define the shape. To position the shape on the screen, you must now tell VDI where to put each vertex. This is done by POKEing horizontal (*x*) and vertical (*y*) coordinate values into a second parameter block area.

The second parameter block begins at a memory location defined by the reserved variable `PTSIN` (*Points Input*). Again, you don't need to know the actual memory locations involved; the computer keeps track of them for you. All you need to do is POKE the correct numbers into `PTSIN` (and even-numbered adjacent locations, in some cases).

Lines 50–80 of the program perform this job by POKEing the bar's *x* and *y* coordinates into memory. The *x* coordinate of the first point is POKEd into `PTSIN`; the first point's *y* coordinate goes into `PTSIN+2`. The *x* coordinate of the second point goes into `PTSIN+4`, and so on. Keep in mind that you must supply a *pair* of coordinate values for every point that you define in `CONTRL+2`.

A third parameter block, beginning at the address defined by the reserved variable `INTIN`, is used to pass *attribute val-*

ues, if any are required by the current subroutine. The term *attribute* is a catchall that can include many different parameters—colors, rotation values, a style index, or whatever—depending on which subroutine is called. Since the subroutine used in this program requires no attributes, we don't need to POKE any values into this segment of memory. As a signal to the VDI handler that no attributes are involved, we must also POKE a zero into location `CONTRL+6`. This location tells the system how many attribute values to read from the `INTIN` parameter block.

After all the required values have been POKED into memory, line 90 of the program executes the `VDISYS` command, which calls the VDI routine and draws a bar on the screen. This may seem an enormous amount of preparation for such a simple task (which some other computers can do with a single BASIC statement). On the other hand, it's better than not being able to draw a bar at all. You can cut down on the bulkiness of the code by writing setup subroutines that contain all the necessary overhead.

### General Drawing Routine

Though every VDI call requires several preparatory steps, each individual step is easy to perform. It should be apparent by now that there's nothing mystical about the process. All you need to do is leave the right pieces of information in places where the computer can find them; then signal that you want the job done. The system itself does the real work.

Though the general procedure is the same in every case, each VDI routine requires different types and amounts of information. One of the most useful VDI routines is the generalized drawing primitive in Program 1. Table 1 summarizes the POKes you need to call this routine.

**Table 1. Generalized Drawing Primitive**

POKE <code>CONTRL</code> , 11
POKE <code>CONTRL+2</code> , number of vertices
POKE <code>CONTRL+6</code> , number of attributes
POKE <code>CONTRL+10</code> , subfunction number (primitive ID)



Again, CONTRL receives the opcode number of the VDI routine; CONTRL+2 the number of vertices in the desired shape; CONTRL+6 the number of attributes (if any); and CONTRL+10 the primitive ID for the subroutine you want. This particular VDI routine is extremely versatile. It can draw pie-shaped segments, ellipses, filled or empty rounded rectangles, and other graphics images, including text. Table 2 lists the primitive IDs for each of this routine's subroutines.

**Table 2. Drawing Subroutines**

Primitive	
ID	Subroutine
1	Bar
2	Circle
3	Arc
4	Pie
5	Ellipse
6	Elliptical arc
7	Elliptical pie
8	Rounded rectangle
9	Filled rounded rectangle
10	Justified graphics text

To select a specific subroutine, find its primitive ID in the left column of Table 2; then POKE that value into location CONTRL+10. Table 3 summarizes the POKES needed to set up the second and third parameter blocks (PTSIN and INTIN). Remember, the value POKEd into CONTRL+2 (number of vertices) determines how many (*x,y*) coordinate pairs you must POKE into the PTSIN parameter block. The *x* and *y* coordinates for the first vertex go into PTSIN and PTSIN+2; the second (*x,y*) coordinate pair goes into PTSIN+4 and PTSIN+6; and so forth.

To draw a circle, ellipse, arc, or pie-shaped segment, POKE *x* and *y* coordinates for the shape's center point into PTSIN and PTSIN+2. A simple circle requires a radius value in PTSIN+8. Arcs and pie shapes built from a part of a circle require a radius value in PTSIN+12. To draw an ellipse, or an arc or pie shape built from part of an ellipse, POKE the shape's *x* radius into PTSIN+4 and its *y* radius into PTSIN+6.

**Table 3. PTSIN and INTIN Parameter Blocks**

POKE PTSIN	<i>x</i> coordinate of first vertex (rectangle) <i>x</i> coordinate of center (circle, ellipse)
POKE PTSIN+2	<i>y</i> coordinate of first vertex (rectangle) <i>y</i> coordinate of center (circle, ellipse)
POKE PTSIN+4	<i>x</i> coordinate of second vertex (rectangle) <i>x</i> radius for ellipse
POKE PTSIN+6	<i>y</i> coordinate of second vertex (rectangle)
POKE PTSIN+8	Radius (circle only)
POKE PTSIN+12	Radius (circular arc or pie only)
POKE INTIN	Start angle for arcs and pies
POKE INTIN+2	End angle for arcs and pies

Most of these subfunctions don't require any attribute values. To draw arcs or pie shapes, however, you must POKE two attribute values into INTIN and INTIN+2 to define starting and ending angles, respectively. Since the angle values are specified in tenths of a degree, not in whole degrees, these parameters can range from 0 to 3600. The starting angle specifies where you want the rounded portion of the arc or pie segment to begin, and the ending angle shows where that portion should stop. The statement POKE CONTRL+6,2 signals that you're passing two attribute values to the VDI.

As you'll learn from experimenting with these routines, VDISYS opens the gateway to a large variety of graphics capabilities. Once you become familiar with the setup process, you'll probably find yourself using VDISYS more and more.

## VDISYS and the Mouse

If you've ever tried to read the ST's mouse controller from BASIC, you know that BASIC lacks commands to read the mouse position or button status. Like certain other tasks, mouse reading can be done from BASIC only with the aid of



VDISYS. Once you know how to read the mouse, you may also want to change the mouse pointer's appearance.

### Entering ST Programs

Before you type in Program 2, "ST Mouse Pointer Editor," here are some tips that will make it easier to enter ST BASIC programs. First, although it may be obvious, it is far easier to enter a program from the Edit window than from the Command window. (To move to the Edit window, type EDIT at the Command window's OK prompt, or choose the Start Edit option from the Edit menu.) The Edit window's full-screen editor is much more convenient for entering program lines than is the Command window's single-line interface. You can also run a program directly from the Edit window (type RUN or choose the Start option from the Run menu). When the program is finished, control returns to the Edit window, so you can immediately modify or add new lines to the program.

The Edit window has one feature that you may or may not appreciate. Until you press Return, the line you're working on will appear in *ghost mode* (the letters will look gray and fuzzy). The purpose of ghost mode is to show which lines you have changed. This is helpful to inexperienced programmers, but, since ghosted letters are harder to read than normal ones, it can be an annoyance. To disable ghost mode, enter this line in the Command window:

```
POKE SYSTAB+2,0
```

Another way to ease the task of program entry is to increase the speed of the cursor. This is done from the Control Panel. The second slider from the top (the one with a rabbit and a turtle) controls the cursor speed. To increase the speed, click on the slider and drag it to the left (toward the rabbit). To slow it down, drag the slider to the right. You can also turn the keyboard beeping sound off and on by clicking the C key icon in the Control Panel.

### Redesigning the Pointer

You need to do two more things before typing in the pointer-editing program. First, set the computer to medium resolution (use the Set Preferences option). Second, turn off buffered graphics from BASIC's Run menu. If your ST has more than 512K of random access memory (RAM) or the TOS operating

system in read only memory (ROM), the second step may or may not be necessary. But in any case, it can't hurt.

Now enter Program 2 and save it to disk. It lets you change the mouse pointer from the familiar arrow shape to a custom design of your own. When you run the program, a grid appears on the left side of the screen, and the word *DONE* is shown on the right. To edit the pointer shape, move the mouse pointer into the grid. Then click the button on any square you want to change. Clicking on a square toggles it on or off. If the square is on (dark) when you click, it will be turned off (erased) and vice versa.

### Program 2. ST Mouse Pointer Editor

```
10 fullw 2:clearw 2:color 1,1,1,1
20 dim ar(16,16),shape(30):for a=1 to 16:for b=
  1 to 16:ar(a,b)=0:next b,a
30 for a=50 to 306 step 16:linef a,20,a,148:nex
  t
40 for a=20 to 148 step 8:linef 50,a,306,a:next
50 gotoxy 48,8:print "done":q=430:r=72:linef q,
  r,q+36,r:linef q+36,r,q+36,r+10
60 linef q+36,r+10,q,r+10:linef q,r+10,q,r
70 gosub readmouse:if lbutton=toggle then 70 el
  se toggle=lbutton
80 if lbutton then gosub flip:goto 70 else goto
  70
90 readmouse: poke contrl,124
100 poke contrl+2,0:poke contrl+6,0:vdisys (0)
110 x=peek(ptsout):y=peek(ptsout+2)
120 lbutton=peek(intout):rbutton=peek(intout+2)
130 return
140 flip:gosub locate
150 if xp>24 and xp<27 and yp=8 then goto define
  it
160 if xp<1 or xp>16 or yp<1 or yp>16 then retur
  n
170 if ar(xp,yp) then ar(xp,yp)=0:color 1,0:goto
  190
180 ar(xp,yp)=1:color 1,2
190 fill xp*16+44,yp*8+18:return
200 defineit: gotoxy 43,12:print "choose hot spo
  t":gosub hotspot
210 poke contrl,111:poke contrl+2,0
220 poke contrl+6,37:hx=xp:hy=yp
230 poke intin,xp-1:poke intin+2,yp-1:poke intin
  +4,1
240 poke intin+6,0:poke intin+8,1
```



## CHAPTER FOUR

---

```
250   for a=10 to 40 step 2:t=0
260   for b=16 to 1 step-1:t=t-2^(16-b)*(ar(b,a/2-
      4)=1):next
270   poke intin+a,t:poke intin+a+32,t:shape((a-10
      )/2)=t
280   next:vdisys (0):clearw 2:gotoxy 32,0:print "
      defined"
290   print "Do you want to save this shape?":a=in
      p(2):if a<>121 then end
300   input "filename";fn$:open "0",#1,fn$:print #
      1,hs;hy
310   for a=0 to 15:print #1,shape(a):next:close #
      1:end
320   hotspot: gosub readmouse:if lbutton=toggle t
      hen 320 else toggle=lbutton
330   if lbutton=0 then 320
340   gosub locate:if xp<1 or xp>16 or yp<1 or yp>
      16 then 340 else return
350   locate: xp=int((x-50)/16)+1:yp=int((y-40)/8)
      +1:return
```

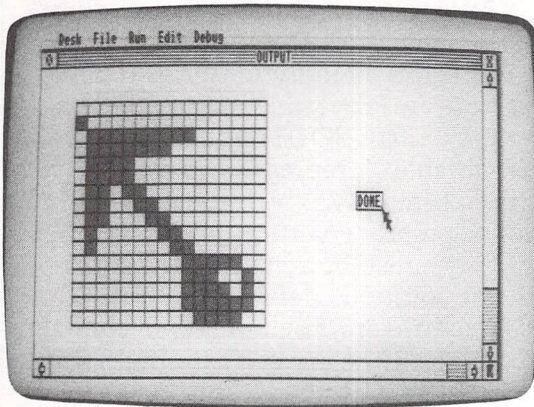
Once you're satisfied with the new pointer, move the mouse out of the grid and click on the word *DONE*. The program will ask for the location of the new pointer's *hot spot*. The hot spot is a single dot that the computer uses to tell exactly what the pointer is pointing at. On the regular mouse pointer, the hot spot is located at the very tip of the arrow. But you can place it anywhere within your custom pointer shape.

After you locate the hot spot, the new pointer appears on the screen. At this point, the program asks whether you want to save the pointer shape data to a disk file for later use. If you do, press Y and enter a filename when prompted. If you press any other key, the program ends without saving the shape. Program 3, "Pointer Loader," at the end of this article, provides a method for reloading the shape data from the disk file and making the custom pointer appear in a BASIC program of your own.

### Reading the Mouse

As we mentioned earlier, ST BASIC has no commands to read the mouse or the state of the mouse buttons directly. Fortunately, there is a VDI routine (appropriately named *Readmouse*) which gives this information. It takes only three steps to call this routine. Since *Readmouse* has an opcode of 124, we first execute *POKE CONTRL,124* to tell the ST which VDI routine

## BASIC Programming



*With this drawing grid (created by Program 2), you can create custom mouse pointers for use in your own ST BASIC programs.*

to call (line 90 in Program 2). This routine doesn't involve any vertices or attributes, so `CTRL+2` and `CTRL+6` are POKed with zeros. Once that brief preparation is complete, the statement `VDISYS(0)` actually calls the routine.

Earlier in this article we explained how to pass information from BASIC to a VDI drawing routine. When that routine has finished with its work (drawing a graphic shape), we don't care whether it passed any information back in the other direction. But many VDI routines pass significant information back to BASIC. Thus, calling a routine like `Readmouse` involves a two-way information transfer. You must supply certain data before calling the routine, and when it returns control to BASIC, the routine sends other information back to you.

Parameter blocks named `PTSIN` and `INTIN` are used to pass data from BASIC to a VDI routine. These parameter blocks are paralleled by `PTSOUT` and `INTOUT`, which perform the same operations in reverse. Though they're considered reserved variables (which you can use only in certain, predefined ways), `PTSOUT` and `INTOUT` each point to a block of special storage locations in memory called a *parameter block*. Like `PTSIN`, `PTSOUT` points to a temporary holding area for information about *x* and *y* position coordinates. Like the `INTIN` parameter block, `INTOUT` defines the area where other information (such as attribute data) is passed.



### Position and Button Status

To read the mouse pointer's screen location, call the Readmouse routine and PEEK the memory locations defined by PTSOUT and PTSOUT+2. In Program 2, this is done at line 110. The statement `X=PEEK(PTSOUT)` transfers the value stored in PTSOUT in the variable X, representing the mouse pointer's horizontal position. Similarly, `Y=PEEK(PTSOUT+2)` makes Y equal to the mouse pointer's vertical position.

To read the status of the mouse buttons, call the Readmouse routine and PEEK the locations defined by INTOUT and INTOUT+2. INTOUT returns information about the left button, and INTOUT+2 tells you the status of the right button. If a button is pressed, the value in the corresponding location is 1; if it's not pressed, the value is 0. Program 2 reads both mouse buttons at line 120. When the left button is pressed, the variable LBUTTON is set to 1; when the right button is pressed, the variable RBUTTON is set to 1. Table 4 outlines the information you need to use Readmouse.

**Table 4. Readmouse Parameters**

<b>Input Parameters</b>	
POKE CONTRL,124	Opcode
POKE CONTRL+2,0	Number of vertices
POKE CONTRL+6,0	Number of attributes
<b>Output Parameters</b>	
PEEK(PTSOUT)	Horizontal mouse position
PEEK(PTSOUT+2)	Vertical mouse position
PEEK(INTOUT)	1 = left button pressed
PEEK(INTOUT+2)	1 = right button pressed

### Customizing the Pointer

Though the ST's familiar arrow pointer is suitable most of the time, occasionally you may want it to look like something else. In a drawing program, for instance, why not reshape the pointer as a pencil or a paintbrush? Once you know how to modify the pointer's appearance, you can make it look like a pointing hand, a musical note, a scientific symbol, or whatever else is needed to give your program a customized look.

The VDI routine that redraws the mouse pointer is called Set Mouse Form, usually abbreviated as SMF. Because the

SMF routine requires a lot of information, its setup procedure is fairly complex. The first step, as always, is to POKE the opcode for the VDI routine into CONTRL. Since the opcode for SMF is 111, Program 2 performs POKE CONTRL,111 at line 210. Next, you must POKE the number of vertices (0, in this case) and the number of attributes (37) into CONTRL+2 and CONTRL+6 (lines 210–220).

The mouse pointer can move anywhere on the screen, so there's no need to provide *x* and *y* coordinates for the shape as a whole. However, you must tell the system where, within that shape, it should put the hot spot. The hot spot's coordinates are defined relative to the upper left corner of the new pointer shape. POKE the *x* coordinate value into INTIN and the *y* coordinate into INTIN+2. At the same time, you should also POKE 1 into INTIN+4 (lines 220–230).

### Who Was That Masked Mouse?

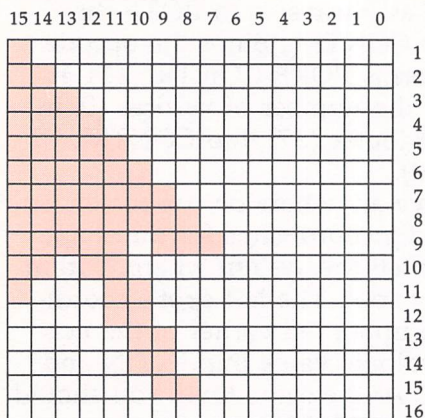
The mouse pointer you see on the screen is made of two separate parts—the pointer shape itself and a second shape called a *mask*. Both forms are the same size (16 pixels high and 16 pixels wide), and they appear at the same place on the screen. Since the pointer and the mask can be different colors, you can make a two-color mouse pointer. To create the illusion of solidity, for instance, you might draw the main body of the pointer in one color and add a darker shadow along its lower edges. To set the mask's color, POKE the desired color number into location INTIN+6. POKE the pointer's color into INTIN+8.

Once you've defined the colors, you must supply shape information for both the pointer and the mask. Each shape requires 32 bytes (16 words) of data. The figure illustrates how the 16 words of shape data go together to make up the entire shape.

If you visualize the pointer shape within a  $16 \times 16$  grid, the first 16-bit data word is in the top row of the grid, the second data word represents the second row, and so on. To pass the shape information to the SMF routine, you must first calculate the 16-bit values represented by the *on* bits within this grid. When that's done, the data for the mask is POKEd into



## Mouse Pointer Data



locations  $\text{INTIN} + 10$ ,  $\text{INTIN} + 12$ , ...,  $\text{INTIN} + 40$ . The pointer shape data is POKEd into locations  $\text{INTIN} + 42$  through  $\text{INTIN} + 72$ . Don't be concerned if that sounds a bit confusing. Program 2 does all the calculations and POKEs for you automatically. For those who are interested, Table 5 outlines the information needed by the SMF routine.

**Table 5. Set Mouse Form (SMF) Parameters**

### Input Parameters

POKE CONTRL,111	Opcode
POKE CONTRL+2,0	Number of input vertices
POKE CONTRL+6,37	Number of attributes
POKE INTIN,X	$x$ = hot spot horizontal coordinate
POKE INTIN+2,Y	$y$ = hot spot vertical coordinate
POKE INTIN+6,mask color	
POKE INTIN+8,pointer color	
POKE INTIN+10-INTIN+40,mask shape data	
POKE INTIN+42-INTIN+72,pointer shape data	

## Saving Custom Pointers

Once you've created a custom pointer with Program 2, it appears on the screen and works just like the regular one. However, as soon as you exit BASIC, the pointer reverts to its usual shape.

To help you incorporate custom pointers into your own BASIC programs, Program 2 lets you save all the pointer shape data in a disk file. Program 3 illustrates how to read the shape data from the disk file and recreate the custom pointer in another program.

### Program 3. Pointer Loader

```
10 dim shape(30):clearw 2:gotoxy 0,0:input "Filename";fn$:open "I",#1,fn$
20 input #1,hx,hy:for a=0 to 15:input #1,shape(a):next
30 poke contrl,111:poke contrl+2,0
40 poke contrl+6,37
50 poke intin,hx-1:poke intin+2,hy-1:poke intin+4,1
60 poke intin+6,0:poke intin+8,1
70 for a=10 to 40 step 2:t=shape((a-10)/2)
80 poke intin+a,t:poke intin+a+32,t
90 next:vdisys (0):clearw 2:gotoxy 0,0:print "defined"
```

The first two data items in the disk file are  $x$  and  $y$  coordinates for the pointer's hot spot. The next 16 data items are the 16 words (32 bytes) of shape data for the pointer and mask forms. After this data has been retrieved, it's simply a matter of performing the setup and calling the SMF routine just as we did in Program 2.

To incorporate this routine into your own program, replace the INPUT statement in line 10 with `FN$="FILENAME"` (using your own filename in place of *FILENAME*). Of course, you could also convert the shape data into DATA statements contained in the program itself.



# File Handling in ST BASIC

Tony Roberts

---

*Until you've learned to use them, files may seem confusing, even dangerous. But a little practice with them makes working with files only slightly more difficult than printing to the screen. Several examples will show you how.*

Word processors, spreadsheets, and database managers are considered powerful programs. They all manipulate data and have the capability of storing the results in files on disk. These files can be recalled later for further refinement or rearrangement.

These programs are powerful because they can interact with information that comes from outside sources—from the disk files. BASIC programmers also have access to this same type of power if they know how to control disk files.

A file is nothing more than a collection of data arranged in such a way that some program will be able to read it. It can be very short, as in a file that keeps track of the high score in an arcade-type game, or very long, as in a database of hundreds of names and addresses.

If you've not yet learned to handle them, files may seem confusing and dangerous. It is possible for you to accidentally erase information on a disk when you're working with files, but it's also easy to learn to avoid such situations. After you've worked with files a few times, the process is only slightly more difficult than printing to the screen. With a little practice and a little thought, your BASIC programs can take on some of the power and elegance of their bigger brothers.

If you've written or typed in programs that use a lot of DATA statements, you've worked with programs that are good candidates for files. Many address-book programs and magazine-article catalogs, for example, keep all associated data in DATA statements. The user of the program is expected to edit the DATA statements as necessary each time the program is run.

When you know all the information that a program will

work with at the time it's written, you'll have no trouble putting that information into DATA statements. However, if you're writing a program that will be used over a long period of time—an address book or a home inventory manager, for example—you don't have all the data on hand. In this case, a disk file that can be edited and updated as time goes by is usually preferable.

Programs that use data files are also more portable, in a way, than programs that rely on DATA statements. Say, for instance, your brother wants to use the home inventory program you have written. Chances are he'll be more comfortable using a program that doesn't require him to break in and edit DATA statements.

### Two Kinds of Files

The two types of files you can access from BASIC are *sequential files* and *random files*. A sequential file is like a letter from a child at camp. You start at its beginning and read straight through until you reach the end. On the other hand, a random file might be compared to a telephone book. Using some kind of index—in this case, the alphabet—you open the book and go more or less directly to the information you need.

Sequential files are adequate for most of the applications a beginning or intermediate programmer will write. In programs that search through massive amounts of information, however, the speed of random access becomes important. Random access files are generally used in tandem with a sequential index file. In other words, the program uses a sequential file to hold an index to the random file. When a record from the random file is needed, the program looks up the information in the index file. The index entry tells the program where to find the complete information in the random file. This method significantly reduces searching time when large amounts of data are involved.

Random files are also handy for those who like to explore disks and study or change the way information is stored. We'll discuss this use of random files in detail a little later.

### Sequential Files

The BASIC keywords OPEN, CLOSE, INPUT, INPUT #, LINE INPUT #, PRINT #, and WRITE # are used to control sequen-



tial files. Program 1, "Score," is a short game program that illustrates how to use these commands to open, close, read from, and write to a file.

### Program 1. Score.

```
20 gosub READFILE
30 gosub PLAYGAME
40 if newscore>score then gosub WRITEFILE
50 end
60 '
70 READFILE:
80 open "I",#1,"score.dat"
90 input #1,score
100 close #1
110 return
120 '
130 PLAYGAME:
140 randomize 0
150 clearw 2
160 print "Previous High: ";score
170 x=int(rnd*26)+97
180 print chr$(x)
190 y=inp(2)
200 if y=x then newscore=newscore+1:goto 170
210 print "Your Score: ";newscore
220 return
230 '
240 WRITEFILE:
250 open "O",#1,"score.dat"
260 print #1,newscore
270 close #1
280 return
```

The game is a simple typing test. The computer selects a random lowercase letter and displays it. You type the letter. The game continues as long as you type each specified letter without an error.

The program begins with a GOSUB to the READFILE routine, which opens and reads the disk file that keeps the high score for the game. The first step in this process is to OPEN the file.

For sequential files, the OPEN command takes three parameters— mode, file number, and filename.

- The *mode* is either an uppercase *I* (input) or an uppercase *O* (output).

- The *file number* is a number, from 1 to 15, which uniquely identifies the file. This number will be used in conjunction with the input and output commands each time you access the file. You cannot have two open files with the same file number, but you can reuse a file number once the first file using that number has been closed.
- The *filename* is the name of the file you want to access.

Line 80 of the Score program OPENS a file named *score.dat* for input and assigns it the file number 1. Line 90 uses the INPUT # command to read some information from the file and store it in the variable *score*. This command operates very much like the INPUT command that requests information from the keyboard, except a file number is included to tell BASIC where to find the information.

The CLOSE command in line 100 closes the file and frees the file number to be used again. Unless the file is reopened, no more information can be read from it.

When a file number is specified with the CLOSE command, only the file associated with that number is closed. If CLOSE is used with no parameters, all open files are CLOSED. Although several other commands—END, RUN, NEW, LOAD, and QUIT among them—cause any open files to close, it's good programming practice to have your program close files as soon as it has finished accessing them.

Closing a file makes sure that all data that was intended to be written to the file is written, and that the system's directory entries for that file are updated if necessary. Problems can occur when a disk with an open file is removed from the drive or if the system is shut down, either at the power switch or via a power failure, while a file is still open. These problems are rare. They can usually be prevented by closing files as soon as access to them is no longer needed.

After the program has read the previous high score from the disk file, it proceeds to the game loop, where it stays until the player makes an error. When the game ends, the program checks to see whether a new high score has been earned. If so, the program reopens the *score.dat* file, this time for output, and uses the PRINT # command to print the variable *newscore* to the file. Note that if a file opened for output does not exist, a new file will be created. On the other hand, if the named file does exist, its previous contents are erased.

When you run Score, you may be surprised to find that it



halts abruptly with a file-not-found error at line 80. This is because BASIC cannot open a nonexistent file for INPUT. One way around this problem is to skip the input section of the program the first time it is run. Run the program using the command

**RUN ,30**

The ,30 tells BASIC to start the program at line 30, which calls the play loop. When the program ends, the *score.dat* file will be created. Next time the game is played, it will run normally from the beginning.

It's a safe bet, however, that most of your data files will have more than one element. This simple example, however, provides a foundation to build on. Try making the following changes to the program.

```
90 input #1,score,name$
155 input "Enter your first name. ",newname$
160 print name$;" had previous high:,"score
260 print #1,newscore,newname$
```

Now the data file will keep both the high score and the name of the player who made that score. When you run the new program, though, it will again stop with an error. Line 90 tries to read two items from the file—*score* and *name\$*—but at this point, the file contains only a single item.

Again, run the program from line 30 to skip the input section and create a new file that matches the current specifications. This points out a key rule in programming with files: The input section must be able to read what the output section has written. There is a similarity here between files and READ and DATA statements. Note the problem that would occur if the variables in line 90 were reversed. Before writing variables to a file, you should have an understanding of how you're going to read them back.

### Another Approach

In Program 1, we knew exactly what we wanted from the file. The program opened the file, read the necessary elements, and then closed the file. When the number of elements in the file is unknown, however, it is necessary to take a different approach, as Program 2, "Birthday List," illustrates.

### Program 2. Birthday List

```
10 gosub INITIALIZE
20 gosub READFILE
30 gosub PRINTLIST
40 gosub ENTERNAMES
50 gosub PRINTLIST
60 if c-1>recordsread then gosub WRITEFILE
70 end
80 '
90 INITIALIZE:
100 clear
110 c=0
120 fullw 2:clearw 2
130 erase name$,age,date$
140 dim name$(50),age(50),date$(50)
150 return
160 '
170 READFILE:
180 on error goto 550
190 open "I",#1,"bdaylist.dat"
200 on error goto 0
210 while not eof(1)
220 c=c+1
230 input #1,name$(c),age(c),date$(c)
240 wend
250 recordsread=c
260 close #1
270 return
280 '
290 ENTERNAMES:
300 c=c+1
310 input "Name: ";name$(c)
320 if name$(c)="END" or name$(c)="end" then 370
330 input "Birthday: ";date$(c)
340 input "Age: ";age(c)
350 print
360 goto 300
370 return
380 '
390 WRITEFILE:
400 open "O",#1,"bdaylist.dat"
410 for i=1 to c-1
420 write #1,name$(i),age(i),date$(i)
430 next
440 close #1
450 return
460 '
470 PRINTLIST:
480 clearw 2
490 for i=1 to c
```



## CHAPTER FOUR

---

```
500   if name$(i)="end" or name$(i)="END" then 520
510   print name$(i),date$(i),age(i)
520   next
530   return
540   '
550   ERRTRAP:
560   if err=53 then resume 270
570   print "Error #";err;"in line";erl
580   end
```

Program 2 lets you enter the names, birthdays, and ages of your friends. Then it saves the information you entered to a disk file. Later, you can run the program again to view the data or to add more names. When you're finished entering data, type END to end the program.

For each name you enter, the program saves two string variables—*name\$* and *date\$*—and one numeric variable—*age*. But the number of names in the list is undefined. In order to know how much to read, the program uses the EOF (end of file) function to detect the end of the file. The statement **while not eof(1)**

in line 210 sets up a loop that reads the three variables for each name until there's no more data left in the file. When the end of a file has been reached, EOF becomes true. The number in parentheses after EOF is the file number of the file you're reading.

In line 230 of this example, the order in which the variables appear is significant. Variables must be read or input in the same order in which they have been written. It's possible to read a numeric variable into a string, but an attempt to read a string into a numeric variable will result in an error.

Since Program 2 begins, as did Program 1, by reading a file from the disk, you might expect it to halt with a file-not-found error the first time it is run. In this case, however, we've allowed for that problem by setting up an error trap in line 180. The ON ERROR command in that line causes program control to be diverted to line 550 if an error, such as a file-not-found error, is detected.

As this routine is written, it checks to see whether the error was error number 53 (File not found on disk drive specified). If that's the case, the program is told to RESUME at line 270, which exits the READFILE routine and continues with the

rest of the program. If any other error is encountered, the program will end.

The statement

**on error goto 0**

in line 200 turns off the special error trapping and returns responsibility for error handling to BASIC.

### Trapping Errors

There's a lot of potential for errors when you're working with files. Nothing is more frustrating than entering a lot of data and having the program crash. A good program identifies likely spots for errors, traps those that occur, and provides a graceful way to continue the program.

When you set up error-trapping routines, there are a couple of things to remember. First, the **ON ERROR** statement must direct flow to a *line number*, not to a *line label*. Second, when you use the **RENUM** command to renumber your program, you'll have to edit any line numbers after **RESUME** in your routine. **RENUM** fails to adjust line number references after **RESUMES**.

After you finish adding names to the birthday list, the program reopens the data file, this time for output, and uses a loop to write all the data in the *name\$*, *date\$*, and *age* arrays to the disk. Opening a file for output erases the contents of any file on the disk with the same name.

When you're working with sequential files, you must follow the example of this program and open the file for input, read all the information into memory, add to the information, edit it or sort it, open the file for output, and write all the information back to disk. It is not possible simply to append new information to the end of an existing sequential file.

The **WRITE #** command in line 420 writes the data out to disk. It is much more convenient to use than the **PRINT #** command used in the previous program.

**PRINT #** works in the same way that **PRINT** or **LPRINT** works, except the information is sent to the disk. If you separate the variables with semicolons, no space will be left between the values on the disk. If you separate the variables with commas, spaces will be inserted after the first value to move the data to what would be the next tab position. Although this approach worked fine in the *Score* program, it can



present problems the next time the data is read with an INPUT # command.

WRITE #, on the other hand, is designed with INPUT # in mind. As it prints to disk, WRITE # encloses strings in quotation marks and prints commas between each value. This makes inputting the information later much easier. When using BASIC dialects that have no WRITE # command, programmers use the PRINT # command, but they must take responsibility for keeping data items separate. The following line writes the same information to disk as does line 420 of the Birthday List program:

```
print #1,chr$(34); name$(i); chr$(34); ","; age$(i); ",";  
chr$(34); date$(i); ","; chr$(34)
```

### Seeing Is Believing

Seeing the information as it's stored on disk will give you a better understanding of what your WRITE # or PRINT # statement is doing. This will help you set up your programs so that data can be retrieved without error.

Program 3, "File Read," allows you to read through data files sequentially to see how the information is stored. The program provides two ways to view the data—character by character or line by line. You'll probably find the character method too slow to be useful, but it is included for the sake of discussion.

### Program 3. File Read

```
10 gosub INITIALIZE  
20 if type=49 then gosub CHARREAD else gosub LINEREAD  
30 end  
40 '  
50 INITIALIZE:  
60 fullw 2: clearw 2  
70 line input;"Please enter a filename. ";filename$  
80 Print "Read file by: 1. Character"  
90 print space$(14);"2. Line"  
100 type=inp(2)  
110 if type=49 or type=50 then return  
120 print chr$(7);:goto 100  
130 '  
140 CHARREAD:  
150 open "I",#1,filename$
```

```
160 while not eof(1)
170 a$=input$(1,#1)
180 if asc(a$)=13 then print chr$(13):goto 200
190 print a$;
200 wend
210 close #1
220 return
230 '
240 LINEREAD:
250 open "I",1,filename$
260 on error goto 340
270 line input #1,a$
280 print a$
290 goto 270
300 close #1
310 on error goto 0
320 return
330 '
340 ERRTRAP:
350 if err = 62 then resume 300
360 print "Error #";err;"in line";erl
370 end
```

The program uses the INPUT\$ command to accomplish the character-by-character display of the file. Line 170 sets a\$ equal to one character read from file #1. The program then prints a\$ and goes on to get the next character. If line 180 is omitted from the program, carriage returns are ignored, and characters are printed, one after another, filling the lines. When INPUT\$ finds a carriage return character, it apparently tosses it out before printing it. Line 180 skirts this limitation by identifying when a carriage return is read and forcing it to be printed.

The CHARREAD routine uses the *while not eof(1)* construct to control the loop that plucks the data from the disk. Since the information is being read a character at a time, the end of the file can be detected without error, and the loop can be exited gracefully.

The LINEREAD routine, on the other hand, presents a different problem. This routine uses LINE INPUT # to fetch data one line at a time. LINE INPUT # can read strings of up to 254 characters. It gathers data from the disk until it reaches a carriage return (ASCII 13, hex 0D) not immediately preceded by a linefeed (ASCII 10, hex 0A). The next LINE INPUT # statement picks up where the previous one left off.



In most cases, though, when a carriage return is stored in a disk file, it is stored as a carriage return/linefeed combination. `LINE INPUT #` reads through the carriage return character, but leaves the linefeed character for the next `LINE INPUT #` to pick up.

What happens in this example is that `LINE INPUT #` reads lines from the disk until the final carriage return is reached. At that point, there's a linefeed character left to be read. The next `LINE INPUT #` reads the linefeed, but it continues reading, looking for a carriage return. Since there is no carriage return, the program halts on an error 62, indicating that the end of the file has been reached.

Having anticipated this problem, however, the `LINEREAD` routine has set an error trap to detect this condition and exit from the program smoothly.

The File Read program can help you see how information is stored in your data files, but it can't show you everything. Try using this program to view `BASIC.PRG`, and you'll be disappointed. If you choose the character read option, you'll see one character before the program ends; the line read option will show you nothing.

The reason for this is that the second character in the `BASIC.PRG` file is an ASCII 26 (hex 1A), also known as Control-Z. With sequential files, Control-Z signals the end—you can read no further. To get the clearest picture of how data is stored on the disk, we'll have to turn our attention to random files.

### Random Files

Random files most often are used in applications where portions of large databases are manipulated, sorted, edited, and printed. The Department of Motor Vehicles in your state, for example, probably uses a random file system to store information about you and your automobiles. Given an index item, such as your license tag number, the program can quickly display all information pertaining to you. This information can be corrected if necessary and quickly rewritten to the master file without your having to read in and rewrite every other bit of data in the file.

This capability results in great efficiency for businesses or governmental agencies with huge amounts of data to manipulate, but for home programmers and their typically smaller

databases, the additional work required to set up a random filing system is probably more than it's worth. If your application is sophisticated enough that random access files would be of benefit, you should probably consider writing the program in a language other than BASIC, as well.

For those reasons, the following discussion of random files is somewhat abbreviated. It focuses on how random files can be used in small utility programs rather than on the intricacies of setting up a full-blown database with random files.

Program 4, "Random Read," explores in detail any disk file, providing you with complete information about how data is stored. Such detailed information is essential if you need to convert data from one format to another. This can be useful when you want to use the data files of one program with another program. A common example is when someone who has just purchased a new word processor wants to use files created by an old program without the labor of retyping. By examining the structure of the data files of each of the word processors, you can determine what conversion needs to be done to make the files compatible.

Although Random Read contains no code that writes information to the disk, it is appropriate here to recommend that whenever you're exploring and experimenting with disk files you do so with backup copies. *Do not experiment with your only copy of a valuable document or program.* Enough said.

### Program 4. Random Read

```
10 gosub SETUP
20 open "R",#1,filename$,128
30 field #1, 128 as a$
40 numrecs=lof(1)
50 if offset>0 then gosub FIXOFFSET
60 while loc(1)<numrecs
70 get #1
80 gosub NEXTRECORD
90 gosub PRINTRECORD
100 wend
110 close #1
120 end
130 '
140 FIXOFFSET:
150 record=offset!/128
160 offset!=offset!-(offset! mod 128)
170 get #1,record-1
180 return
```



## CHAPTER FOUR

```
190 '
200 PRINTRECORD:
210 for i=0 to 127 step 16
220 print using "#####";offset!;
230 print ":";
240 for j=1 to 16
250 b$(j)=mid$(a$,i+j,1)
260 hx$=hex$(asc(b$(j)))
270 if len(hx$)=1 then hx$="0"+hx$
280 if j=8 then form$="\ \! " else form$="\ \"
290 print using form$;hx$;
300 next j
310 print space$(2);
320 for k=1 to 16
330 if asc(b$(k))>32 then print b$(k); else prin
t ".";
340 next k
350 offset!=offset!+16
360 print
370 next i
380 return
390 '
400 NEXTRECORD:
410 if first=0 then first=1:goto 460
420 print:print "Press a key to continue, Escape
to quit."
430 dummy=inp(2)
440 if dummy=27 then close #1:end
450 gotoxy 0,12: print space$(40)
460 gotoxy 0,1
470 print "Displaying record #";loc(1);"of";numr
ecs
480 print
490 return
500 '
510 SETUP:
520 dim b$(16)
530 fullw 2:clearw 2
540 width 80
550 first=0
560 gotoxy 0,0
570 input "Specify a filename. ",filename$
580 input "Specify an offset. ",offset!
590 return
```

Random files are opened in much the same way as sequential files. The differences are these: An uppercase *R* is used to indicate the file mode; a fourth parameter, *record*

*length*, is added to the open statement; and a FIELD statement is used to describe how the data within the record is to be used.

There also are significant differences in the way data is moved to and from the disk. In general terms, here's what happens. When a file is opened, BASIC sets aside a *buffer* in memory to act as a way station between the disk file and the program. When information is requested from the disk, it is read a record at a time and placed in the buffer where the program can use it. Going from program to disk is similar. A record's worth of data is placed in the buffer; then the buffer contents are written to disk. The GET # command is used to move data from disk to buffer. PUT # writes data from the buffer to the disk.

In line 20 of Program 4, the final parameter, 128, is the record length. Record length can range from 1 to 4096. If no length is specified, it defaults to 128 bytes. In line 30, the FIELD statement describes how the data in the record is to be used. In this program, the entire buffer, 128 bytes, is assigned to the buffer variable a\$. A field statement such as

**field #1, 64 as a\$, 64 as b\$**

assigns the first half of the buffer to a\$ and the latter half to b\$.

Now that the file is open and fielded, line 40 uses the LOF (length of file) function to find out how long the file is in *records*, not in bytes as the BASIC manual indicates. You can find the number of bytes in a random file by specifying a record length of 1 in the OPEN statement. This is accurate only as long as the file contains fewer than 65536 bytes. After that, the counter rolls over like the odometer on an aging automobile.

At any rate, line 40 discovers the number of records in the target file and assigns that value to the variable numrecs. The comparison in line 60 employs the LOC function to make sure the end of the file has not been reached, but tests the current record number—as supplied by LOC—against the value stored in numrecs.

When the the program runs, you are asked to specify a filename and an offset. If the offset is between 0 and 127, the file display starts with the first record. If it's between 128 and 255, the display begins with the second record, and so on.

The format for the display in this program is much like that commonly found with disk editor programs. Each record



is displayed on eight screen lines. First you see the offset, then the hexadecimal values for the first 16 bytes of the block. The line is completed by an ASCII representation of those same 16 bytes.

The program allows you to step through a file, a block at a time, viewing the data. If you want to move to a distant section of the file, quit the program and rerun it specifying the desired offset. If an offset greater than zero has been specified, the program uses the full form of the GET # command to move to the start of the desired record.

The optional second parameter of GET # is the record number that is to be read. If the second parameter is not specified with GET #, the next available record is read. The FIX-OFFSET routine identifies the appropriate record number; then it reads the previous record. This places the file pointer on the desired record, so the next time the GET # statement in line 70 is executed, the proper record is read.

Once again, this program does not allow you to change any data in the disk file (as would be possible with a commercial disk editor program). Modifications could be made to permit this capability, but they are beyond the scope of this discussion.

Another deficiency of this program is that it will not display an incomplete record at the end of a file. Say, for example, that you have a file that measures 150 bytes. In measuring the number of records in the file (line 40), the LOF function returns the number of *complete* records. In this case, the function returns a 1, for the first 128 bytes. The remaining 22 bytes are not accessible by the program.

### Altering Data

Once you've studied a file with Random Read, you can use a program such as Program 5, "Conversion Example," to alter the data throughout the file. I've used this program to make *STWriter* files compatible with other word processors.

*STWriter*, which shipped with early versions of the ST, stores carriage returns on disk as ASCII 0. This unusual arrangement made it a real nightmare to use *STWriter* files with other word processors or for telecommunications. Studying the *STWriter* files to learn how they were stored, however, made it relatively simple to modify them for other uses.

### Program 5. Conversion Example

```
10 gosub SETUP
20 gosub OPENFILES
30 gosub COMPARE
40 close
50 end
60 '
70 COMPARE:
80 while not eof(1)
90 get #1
100 if asc(a$)=0 then print #2,crlf$; else print
    #2,a$;
110 gotoxy 1,2
120 print using "#####";counter
130 counter=counter-1
140 wend
150 return
160 '
170 OPENFILES:
180 open "R",#1,infile$,1
190 field #1,1 as a$
200 counter=lof(1)-1
210 open "O",#2,outfile$
220 gotoxy 1,2
230 print using "#####";counter;
240 print " bytes to convert."
250 return
260 '
270 SETUP:
280 fullw 2:clearw 2
290 gotoxy 0,5
300 crlf$=chr$(13)+chr$(10)
310 input "File to convert.      ",infile$
320 input "Name for new file.    ",outfile$
330 if infile$<>outfile$ then 370
340 clearw 2
350 print chr$(7);"* * * File names must differ
    * * *"
360 goto 290
370 return
```

Conversion Example opens the source file (#1) as a random access file with a record length of one byte. At the same time, the program opens a destination file (#2) as a sequential file. Line 90 GETs a byte from file #1. Then line 100 checks to see whether that byte is ASCII 0. If not, the character is simply printed to file #2, and the program continues. If the char-



acter is an ASCII 0, a carriage return/linefeed combination is printed to file #2.

Those who purchased their STs a little later probably received the *1ST Word* word processor rather than *STWriter*. Although *1ST Word* has a lot more going for it than its predecessor does, it's not without quirks. For example, normal space characters are stored as ASCII 30 rather than the usual ASCII 32. Another example is that *1ST Word* uses ASCII 32 as a "fixed space" character. The result is that if you import a file from an outside source—say, from a telecommunications service—and attempt to edit it, you'll be confounded by the incompatibility of space characters.

The solution: Edit line 100 of Conversion Example to check for ASCII 32 and to print an ASCII 30 to the output file when it finds one. The new line might look like this:

```
100 if asc(a$)=32 then print #2 chr$(30); else print #2,a$
```

With a couple of other quick modifications, you could use this program to search for and strip printer codes out of a file or to alter those codes for use with another printer. Although the program is relatively slow, it shows that an understanding of files and the use of a short BASIC program can solve a data file problem and save you quite a bit of time.

Because BASIC limits the maximum number of records in a random file to 32767, you should not attempt to use this technique on files longer than 32767 bytes.

### Worth the Effort

Although it takes a little study and a little practice to master file-handling commands, it's certainly time well spent. The application of disk files at the proper times can improve the usefulness and appearance of your programs.

As you can see from the examples, a program need be neither long nor complex to make use of disk files. In fact, once you've grown comfortable with disk file handling, you'll probably find that some of the most useful programs you write will be very short routines that use disk files in one way or another.

# Using GEMSYS and VDISYS in ST BASIC

Philip I. Nelson

---

*To beginning programmers, GEMSYS and VDISYS are among the most mysterious and forbidding commands in ST BASIC. The names themselves suggest connections with GEM and VDI, two important components in the conglomerate of hardware and software that makes up the Atari ST computer. But what exactly can you do with GEMSYS and VDISYS?*

Both GEMSYS and VDISYS are designed to give BASIC programmers access to system routines. These routines, actually machine language programs located in ROM, allow you to perform tasks that would be difficult or impossible with ordinary BASIC commands. The VDISYS statement permits you to call VDI (Virtual Device Interface) routines. The VDI contains dozens of low-level routines that can do many different tasks. These routines draw circles, ellipses, and boxes; manipulate raster blocks; change character fonts; and read the screen location of the mouse pointer. The GEMSYS statement serves a similar function for an AES (Application Environment Services) routine. AES routines are considered high-level routines. In this context, low-level simply means that the routine performs a single, comparatively simple task such as drawing a line. High-level means that the routine performs a more complex job such as displaying a dialog box and retrieving input from the user. Thus, AES contains routines to control such GEM features as windows, menus, and dialog boxes.

## **Accessing VDI and AES Routines**

Both GEMSYS and VDISYS take a form which is technically that of a BASIC function. That is, each ends with a pair of parentheses, inside which you supply some information. But they're really special-purpose commands unique to the ST.



Like the CALL statement, their main purpose is to call a machine language routine located somewhere in the computer's memory. However, CALL is designed primarily for user-written ML routines. It requires that you know the actual memory address where the routine begins. GEMSYS and VDISYS work only with system routines and don't require that you know where the system routine is located. The ST finds the routine for you automatically and handles several other details as well.

Virtually every GEMSYS and VDISYS call requires a certain amount of preparation. If you try to use either command without the setup that it requires, the result is often a system crash. To regain control, you must either press the Reset switch or turn the computer off and on again. The ST is not a forgiving computer where system routines are involved, and this emphasizes an important fact: Like other system software, AES and VDI routines are designed first and foremost for the computer's own use. GEMSYS and VDISYS provide a "back door" through which you can use them from BASIC, but none of these routines was designed with BASIC in mind. The bulk of the ST's operating system was written in the C language, which uses conventions different from BASIC to pass information from one program to another. The setup procedures required for these routines are quite convenient in a C program, but cumbersome in BASIC.

### Parameter Blocks

Every AES and VDI routine expects to receive certain information from you, the programmer, when it is called. The amount and character of the information varies from one routine to the next. It may be as simple as the number of an error message you wish to display in a dialog box, or as complex as a set of coordinates and vertices for a graphics drawing routine. In every case, the information is passed through known memory locations, which the system itself monitors for you. These locations—actually memory zones of a predefined size—are known as parameter blocks.

Let's look at the entry points for the various parameter blocks. Enter the following line in the BASIC Command window and press Return:

**PRINT contrl, intin, intout, ptsin, ptsout, gb**

Even if you've never defined these variables, the ST prints six numbers, indicating that it has already assigned values to them. ST BASIC treats the variables CONTRL, INTIN, INTOUT, PTSIN, PTSOUT, and GB as reserved, meaning they can be used only for a special purpose. An error occurs if you try to redefine their values.

Each of these six variables points to a different parameter block used to access a VDI or AES routine. The first five—CONTRL, INTIN, INTOUT, PTSIN, and PTSOUT—are used with VDISYS, while the last one—GB—is used with GEMSYS. As a convenience, ST BASIC lets you refer to the various parameter blocks with variable names rather than with absolute numeric values. You don't need to know the actual address where CONTRL begins, for instance. If you POKE a value into CONTRL, the ST automatically finds that parameter block for you.

Program 1 shows how to use parameter blocks and also demonstrates a very useful VDI drawing routine. If you are using a color monitor, switch to medium resolution before running the program.

### Program 1. VDI Drawing Routine

```
100 REM VDI 11, generalized drawing primitive
110 FULLW 2: CLEARW 2
120 POKE CONTRL, 11 : REM Opcode for VDI routine
130 POKE CONTRL+2, 2 : REM Number of vertices for
    shape
140 POKE CONTRL+6, 0 : REM Nothing in INTIN
150 POKE CONTRL+10, 8 : REM Primitive ID for round
    ed rectangle
160 POKE CONTRL+12, 1 : REM Device handle
170 FOR j=50 TO 180 STEP 3
180 POKE PTSIN, j : REM X coord of upper left
    corner
190 POKE PTSIN+2, j : REM Y coord of upper left
    corner
200 POKE PTSIN+4, 200 : REM X coord of lower right
    corner
210 POKE PTSIN+6, j+50 : REM Y coord of lower right
    corner
220 VDISYS(0) : REM Do the deed!
230 NEXT
240 GOTO 110
```



In order to use any AES or VDI routine, you must know the opcode, or identifying number, for that routine. For VDI routines, the opcode is always POKEd into the location defined as CONTRL. In Program 1, that job is performed in line 120, where we execute `POKE CONTRL, 11`. This statement stores the value 11 in the first two-byte segment of the CONTRL parameter block.

CONTRL, as we've seen, always receives the opcode number of the VDI routine we wish to call. In line 130, the next location in the CONTRL area, `CONTRL+2`, receives the number of vertices (corners) required to define the shape to be drawn. Since the shape-drawing routines in VDI routine 11 always involve regular (symmetrical) shapes, this number is easy to determine. A triangle, for example, requires three vertices. A rectangle like the one in Program 1 requires only two opposite vertices.

Many VDI routines require a value in location `CONTRL+6`, line 140, which defines the number of attributes for a particular routine. The term *attributes* is a bit of a catchall that can include anything from a style index to color rotation values. VDI routine 11 requires no attributes, so we simply `POKE CONTRL+6, 0`.

### Primitive ID

The next `POKE` for VDI routine 11 goes into location `CONTRL+10` (line 150). This segment of the CONTRL parameter block is very important for this particular routine, since it tells VDI which of the ten different drawing subroutines you wish to call. ST software developers call this value a *primitive ID*. A *graphics primitive* is a low-level routine that performs a single function such as drawing a line or simple geometric shape. In this case, we want to draw an open rectangle with rounded corners. That's just one of the ten different subroutines accessible within VDI routine 11. Table 1 lists the ID numbers for the subroutines available in this routine.

The values in Table 1 select which shape VDI routine 11 will draw. To invoke one of these subroutines, you must `POKE` the corresponding primitive ID number into location `CONTRL+10`. The rest of the preparation depends on which subroutine you choose. Table 2 summarizes the values to `POKE` into PTSIN and INTIN for all the various routines.

**Table 1. Drawing Subroutines**

Primitive ID	Draws
1	Bar
2	Circle
3	Arc
4	Pie
5	Ellipse
6	Elliptical arc
7	Elliptical pie
8	Rounded rectangle
9	Filled rounded rectangle
10	Justified graphics text

Line 160 places the number representing the device handle in `CONTRL+12`. In this case, the device handle is 1, representing the screen. Sometimes it may be necessary to use another VDI routine to determine the value of the handle. When it is necessary to determine the handle, use VDI routine 100, Open Virtual Work Station.

Most of the remaining preparation for the VDI routine involves setting *x* and *y* coordinates. VDI routine 11 can draw a variety of different graphic shapes; in this case, we're using it to draw a rounded rectangle. Before calling the routine, you must define for VDI the location of four points on the screen—the *x* and *y* location of the rectangle's upper left corner, and the *x* and *y* location of its lower right corner. The parameter block where these points are stored is defined, appropriately enough, by the reserved variable `PTSIN` (PointS INput). Lines 180–210 store the coordinate values in the area beginning at `PTSIN`. Like the `CONTRL` parameter block, the `PTSIN` parameter block is a reserved memory area. Each screen coordinate is treated as a two-byte value; the value of the first coordinate is `POKEd` into `PTSIN`, the second is `POKEd` into `PTSIN+2`, and so forth.

Not all of these locations need to be used in every case. A rectangular shape requires two pairs of *x* and *y* coordinates to define its upper left and lower right corners. A perfect circle requires only two *x* and *y* values to locate its center on the screen; however, you must also supply a radius value in `PTSIN+8` to define its size. The other rounded shapes—an ellipse, arc, or pie segment—require both *x* and *y* radius values.



**Table 2. PTSIN and INTIN Values**

Location	Value
PTSIN	$x$ coordinate of first vertex (rectangle) $x$ coordinate of center (circle, ellipse)
PTSIN+2	$y$ coordinate of first vertex (rectangle) $y$ coordinate of center (circle, ellipse)
PTSIN+4	$x$ coordinate of second vertex (rectangle) $x$ radius for ellipse
PTSIN+6	$y$ coordinate of second vertex (rectangle)
PTSIN+8	Radius (circle only)
PTSIN+12	Radius (circular arc or pie only)
INTIN	Start angle for arcs and pies
INTIN+2	End angle for arcs and pies

## What Goes into INTIN?

Where VDISYS is concerned, PTSIN has a fairly clear function: It stores coordinates or other values for the points needed to create a graphic shape on the screen. But many VDI routines require extra information that doesn't fit into such a neat category. As mentioned above, the INTIN parameter block serves as a pipeline for such miscellaneous bits of data, collectively termed *attributes*. Location CONTRL+6 tells VDI the number of attributes you wish to send. Program 1 stores a zero in this location to signal that subroutine 9 doesn't need any attributes. When attributes are involved, you must be careful to POKE a corresponding number into CONTRL+6 before calling the VDI routine. If you omit this step, the computer will probably crash.

VDI routine 11 requires attributes only when you wish to draw an arc or pie shape. With both of these shapes, you must define starting and ending angles to indicate where the rounded portion of the shape begins and ends. The angle value is defined in tenths of a degree (not in whole degrees). When you draw either of these shapes, you must signal to VDI that two attributes are being passed (with POKE CONTRL+6,2). Then POKE the value of the shape's starting angle into INTIN and its ending angle into INTIN+2. Like CONTRL and PTSIN, the INTIN parameter block is divided into two-byte segments. The first segment is defined by location INTIN; the rest are defined by locations INTIN+2, INTIN+4, and so on.

### Call of the Wild

After the required setup chores have been performed, it's finally safe to call the VDI routine. This is done with the statement `VDISYS(0)`. If you're wondering what the zero inside the parentheses signifies, the answer is nothing. It's a dummy parameter, which means that the ST doesn't care what you put there. You can replace the zero with 123 or any numeric value. The same is not true of `GEMSYS`, however (see below).

Program 1 demonstrates the fundamental mechanics of using `VDISYS` or `GEMSYS` in ST BASIC. First, you store the needed information in memory; then you call the desired system routine. Once you've performed the setup, the computer does the rest of the work. Though the preparatory tasks may seem cumbersome, the computer works very quickly once control has passed to the system routine. It may seem a lot of work, but it's better than not being able to perform the operation at all.

Many system routines do jobs that are otherwise impossible in BASIC. However, not every system routine is useful in the BASIC environment. Some simply duplicate a function that BASIC already provides. For instance, you can draw an open circle with a `VDISYS` call, but that operation offers no real advantage over BASIC's own `CIRCLE` statement. Other system routines don't work because they conflict with BASIC itself. On the other hand, there are certain advantages to working in BASIC. For example, since BASIC has already created an Output window for your use, it's usually not necessary to open a virtual workstation and obtain a device handle before drawing or printing to the screen.

### Two-Way Pipeline

The routine demonstrated in Program 1 involves a one-way information flow—from BASIC to VDI. When the routine is finished, VDI returns information in the form of an error code. But the main object is to pass VDI the information it needs to carry out the appointed task. There are other routines, however, whose purpose is to pass data in the opposite direction—from the system back to BASIC. Program 2 demonstrates one such routine. VDI routine 124 tells you where the mouse pointer is located and which of the mouse buttons is pressed.



### Program 2. Read Mouse Pointer

```
100 REM VDI 124, read mouse pointer
110 POKE CONTRL,124
120 POKE CONTRL+2,0 :REM No vertices
130 POKE CONTRL+6,0 :REM Nothing in INTIN
140 VDISYS(0) :REM vq_mouse
150 x=PEEK(PTSOUT)
160 y=PEEK(PTSOUT+2)
170 button=PEEK(INTOUT)
180 PRINT "x=";x
190 PRINT "y=";y
200 PRINT "button is";
210 IF button=0 THEN PRINT " not";
220 PRINT " pressed"
230 GOTO 110
```

Again, the first step in calling any VDI routine is to POKE the appropriate opcode number (124, in this case) into the location defined by CONTRL. Since it's impossible to pass information of any significance from BASIC to the mouse pointer, you might think that no more preparation is necessary. Why not call the routine immediately with VDISYS? In most cases, you'd get away with this, but it's always prudent to signal to VDI that no vertices or attributes are involved by POKEing zeros into CONTRL+2 and CONTRL+6. Remember, VDI and AES routines aren't intended to be called from BASIC at all. To obtain consistent results, you should make sure that important locations such as CONTRL+2 can't possibly contain left-over values from a previous program or other activity by the user.

### PTSOUT and INTOUT

Program 2 introduces two new variables, PTSOUT and INTOUT. The PTSOUT (PointS OUTput) parameter block is the converse of PTSIN. It serves as an information conduit from the VDI back to your BASIC program. In Program 2, the points returned in PTSOUT are the current *x* and *y* coordinates of the mouse pointer. Similarly, the parameter block beginning at INTOUT (INTEger OUTput) serves as a return area for other information relevant to that routine. To learn what VDI has to say, simply PEEK locations PTSOUT and PTSOUT+2 for the screen coordinates and PEEK location INTOUT for the button status.

In summary, VDISYS involves five separate parameter blocks, each of which begins at the location stored in a reserved variable. The CONTRL parameter block holds the most fundamental data such as the number of the VDI routine you wish to call. The PTSIN and INTIN blocks hold information you need to pass from BASIC to the VDI before calling the routine. And the PTSOUT and INTOUT blocks serve as way stations for information that VDI returns to you. Though the system may seem awkward at first, it does have a logical organization. More to the point, it works.

### Using GEMSYS

The GEMSYS statement calls an AES system routine from BASIC. It works in the same general manner as VDISYS, which we discussed in the preceding section. First, you plant needed information in a memory area where the AES can find it; then you call the desired routine with a GEMSYS command. The AES contains powerful, high-level routines which are generally more complex than VDI routines. Some AES routines, for instance, can create a dialog box, hold an interactive dialog with the user, erase the dialog box and restore the underlying screen, and finally, inform you of the results of the dialog. The price you pay for this increased power is that GEMSYS offers the BASIC programmer even less convenience than VDISYS.

ST BASIC reserves several variables for use with VDISYS. For GEMSYS, only one reserved variable is available—GB. Like the reserved variables CONTRL, PTSIN, INTIN, PTSOUT, and INTOUT, this variable defines the beginning address of a special memory area. Contained in this area are six addresses, each of which points in turn to an AES parameter block.

In other words, the AES, like VDI, communicates with BASIC through a group of parameter blocks. But BASIC does not give you a reserved variable containing the location of each different AES block. Instead, a single reserved variable—GB—points to the beginning of a 24-byte zone that contains the addresses of the six parameter blocks used to access AES routines. Table 3 contains the names usually given to these parameter blocks and the locations used to access them.



**Table 3. AES Parameter Blocks**

Location	Parameter Block
GB	CONTROL
GB+4	GLOBAL
GB+8	GIN TIN
GB+12	GIN TOUT
GB+16	ADDRIN
GB+20	ADDROUT

One of the GB structure's most notable features is that every address is located four bytes past the next. Remember that the contents of these locations are memory addresses. Since the 68000 microprocessor's address space includes 32-bit addresses, we need four bytes to hold the pointer to each parameter block. If you refer to the GB structure with another BASIC variable, use a double-precision variable (A#, and so forth) to prevent address overflow.

Of these six parameter blocks, the last four—GIN TIN, GIN TOUT, ADDRIN, and ADDR OUT—are most commonly used. GIN TIN and GIN TOUT, like INTIN and INTOUT in the case of VDISYS, pass integer values from BASIC to AES and from AES to BASIC, respectively. ADDRIN and ADDR OUT pass addresses back and forth between the two environments. At first it may not be obvious why you would want to pass an address to a routine. The next program in this section will demonstrate a typical use for this technique: After creating a string which an AES routine needs, you pass the address of the string to make it accessible to AES. By telling AES where the string is located, you make it possible for the system routine to use it.

### Why Me?

If this is your first exposure to using AES routines in BASIC, you are probably asking why anyone would design such a diabolically cumbersome scheme for accessing a system routine. The answer, again, lies in the fact that GEM (which includes both VDI and AES) was written in the C language and was designed as a transportable system that could be used on many different computers. In C programs, it is convenient and hence very common to pass information from one routine to

another through control structures such as the six AES parameter blocks. ST BASIC, at least in the incarnation available at the time of this writing, supports only a few, rudimentary GEM features. The most powerful, interactive GEM entities (such as interactive dialog boxes) are completely inaccessible with ordinary BASIC commands. So the BASIC programmer isn't left with much choice. You can either learn to use control structures designed for another language or do without AES features entirely.

### Putting AES to Work

Despite this seemingly complex scheme, some AES routines are quite easy to use. Program 3 demonstrates a very powerful AES routine.

#### Program 3. Form\_Alert

```
100 REM AES 52, form_alert
110 FULLW 2: CLEARW 2
120 a#=GB :REM Get the key to Pandora's box.
130 gintin=PEEK(a#+8) :REM From me to AES.
140 gintout=PEEK(a#+12):REM From AES to me.
150 addrin# =PEEK(a#+16):REM Passes address of text$.
160 defbutn=1 :REM default exit button.
170 POKE gintin,defbutn:REM Press RETURN to choose this one.
180 REM First value in text$ chooses the icon...
190 REM 0=none, 1=NOTE, 2=WAIT, 3=STOP
200 text$="[1][Your message goes here! or here! or here! or even here!]"
210 text$=text$+"[Hi! Boo! Whee...]" +CHR$(0)+CHR$(0)
220 POKE addrin#,VARPTR(text$):REM Tell AES where text$ is.
230 GEMSYS(52) :REM form_alert
240 PRINT "You chose ";
250 ON PEEK(gintout) GOTO Hi, Boo, Whee
260 PRINT "Something is rotten in Denmark":END
270 Hi: PRINT "Hi":END
280 Boo: PRINT "Boo!":END
290 Whee: PRINT "Whee..."
```

Program 3 requires more setup than do the VDI routines illustrated in the previous section, but it does far more than draw a simple graphics shape. Using the AES routine known



## CHAPTER FOUR

---

as `form_alert`, it creates a familiar-looking dialog box which delivers a message to you and invites one of three responses. Dialog boxes of the type shown in Program 3 are familiar to every ST owner, but you may not have realized how easy they are to control from BASIC.

Program 3 begins by storing the address of GB in a double-precision variable named `A#`. Once this location is known, we use `A#` as a reference point to find the addresses of the parameter blocks where we need to store information for the AES routine. Note that we don't care about the actual locations of these memory areas; it's sufficient to refer to them in relative terms, using the addresses we originally PEEKed from the GB control structure. Like the key to Pandora's box, the address in GB is the key that unlocks a host of powerful instrumentalities.

Before we can call the AES routine with `GEMSYS`, it's necessary to POKE two values into memory. The first POKE, in line 170, defines which button in the alert box will be chosen if you exit by pressing Return instead of clicking the mouse. The second POKE, in line 220, is a little trickier. It uses the `VARPTR` function to tell AES the address where the string `TEXT$` begins. `TEXT$` contains all of the text which we'll want the ST to display inside the alert box. When creating your own strings for use with this routine, make sure that they're null terminated—with zero bytes, `CHR$(0)`—as shown in this example.

Once you've stored these two items of information in places where AES can find them, you can call the `form_alert` routine with the statement `GEMSYS(52)`. The value 52 inside parentheses is very significant; it tells the ST which AES routine to execute. Note the difference between the way that `VDISYS` and `GEMSYS` receive the opcode for the routine that you wish to invoke. `VDISYS` ignores whatever you put in its parentheses and expects you to POKE the desired routine's opcode into `CONTRL`. `GEMSYS` expects you to supply the opcode within its parentheses and pays no attention to what's in `CONTRL`.

The ST ordinarily calls the `form_alert` routine when it wants to advise you of a situation that calls for two, or occasionally three, responses. For instance, when you copy or delete a file from the desktop, the ST displays a small alert box indicating the number of files to be copied or deleted. You

may proceed by pressing Return or by clicking the box labeled OK. To cancel the operation, click on the box labeled Cancel.

It's no accident that you're provided with two different ways to select the positive option, but only one way to cancel it. The `form_alert` routine lets you choose which, if any, of the available options is chosen by pressing Return. If you supply a zero, `form_alert` ignores Return and waits until you click one of the displayed boxes. Otherwise, it assigns Return to box 1, 2, or 3, depending on whether you POKE a 1, 2, or 3 into the location defined as GINTIN.

### Form\_Alert Text String

`Form_alert` also expects you to supply a string containing three separate items of information. Note that each item is enclosed in square brackets. The first item in the `form_alert` string is a number from 0 to 3. This value indicates which icon, if any, you want to appear in the left portion of the dialog box. If the icon number equals 0, no icon appears. Values of 1, 2, and 3 are used to select the Note (exclamation point), Wait (question mark), and Stop (stop sign) icons, respectively.

The second portion of the string contains the text message that you want to display in the box. This can consist of as many as five lines of text. Each line is separated from the next by an OR (|) symbol.

The third and final portion of the `form_alert` string contains the text for the option boxes you wish to display within the dialog box. Naturally, you should supply text of some kind for every available option. At the end of the `form_alert` string are two terminating zero bytes. Don't forget to include the terminator bytes. If you forget, the routine may crash.

Once the text string has been defined, you inform AES of its location by POKEing its beginning address into the location defined as ADDRIN. Then comes the GEMSYS call itself. After control returns to BASIC, you will usually want to know which of the available options has been chosen. This information is returned in the location defined by GINTOUT. If GINTOUT equals 0, the routine has failed (this condition is rarely detected from BASIC, since the failure of a system routine frequently involves a total system crash). If GINTOUT equals 1, the first button in the dialog box has been chosen; if it contains the value 2, the second button has been clicked, and so forth.



Program 3 illustrates how to set up this routine and retrieve the information that it returns to BASIC. The dialog box strings may appear trivial, but they point up the fact that you aren't limited to dichotomous yes/no, OK/Cancel situations. Use this dialog routine whenever you wish to present a special message with one, two, or three selectable options.

### Error Boxes

Program 4 illustrates an even simpler sort of dialog box, which is called with AES routine 53.

#### Program 4. Form\_Error Routine

```
100 REM AES 53, form_error
110 FULLW 2: CLEARW 2
120 a#=GB :REM Get the key to Pandora's box.
130 gintin=PEEK(a#+8) :REM From me to AES.
140 gintout=PEEK(a#+12):REM From AES to me.
150 ernum=4 :REM Error # for the error you want
    to display.
160 POKE gintin ,ernum :REM Pass the error # to
    AES.
170 GEMSYS(53) :REM form_error
180 PRINT "You chose ";peek(gintout)
```

The routine used in Program 4 is known as form\_error, and it displays a box containing an error message. The content of the message is determined by the error number you POKE into GINTIN. To view other messages, change the value of ERNUM in line 150 from 4 to some other number. Some errors, such as number 11, include a specific message; others are identified only with a TOS error number.

### Reshaping the Mouse Pointer

If you use the ST for any length of time, you'll probably become familiar with several different pointer shapes. The arrow pointer usually appears in the desktop; when the computer is busy accessing the disk drive or setting up an application, the pointer changes to a busy-bee shape. These shapes, and several others, are selectable under program control with another AES routine known as graf\_mouse. Program 5 shows how to use it.

### Program 5. Reshape the Mouse Pointer

```
100 REM AES 78, graf_mouse
110 FULLW 2: CLEARW 2
120 PRINT "Which mouse pointer shape do you want
    ?"
130 PRINT "0 = Ye Olde Arrow"
140 PRINT "1 = I-beam cursor"
150 PRINT "2 = Busy bumblebee"
160 PRINT "3 = Pointing hand"
170 PRINT "4 = Grabbing hand"
180 PRINT "5 = Skinny crosshair"
190 PRINT "6 = Chubby crosshair"
200 PRINT "7 = Outlined crosshair"
210 PRINT "Enter a number from 0-7 (9 to quit)"
220 INPUT a$: IF a$="9" THEN END
230 IF LEN(a$)<>1 or a$<"0" or a$>"7" THEN GOTO
    120
240 a#=GB : REM Get key to Pandora's box
250 gintin =PEEK(a#+8) :REM From me to AES
260 gintout=PEEK(a#+12) :REM From AES to me
270 POKE gintin, VAL(a$):REM gr_monumber
280 GEMSYS(78) :REM graf_mouse
290 IF PEEK(gintout)=0 THEN PRINT "Bombed":END
300 PRINT:PRINT "So move the pointer already....
    "
310 PRINT:GOTO 120
```

The graf\_mouse routine, AES 78, lets you choose from eight different predefined pointer shapes according to the needs of the moment. In addition to the arrow and the bee, you can choose an I-beam cursor shape, a pointing hand, a grabbing hand, and three different crosshair shapes. After POKEing the desired value into GINTIN, we call the routine with GEMSYS(78). BASIC imposes its own constraints on the pointer, making it invisible whenever you press a key. Move the mouse to make the pointer reappear in its new form.

### Growing and Shrinking Boxes

Another aspect of the AES has to do with the management of windows and menus. Some of these are superfluous to this discussion, since ST BASIC already contains commands to open windows, clear them, and so forth. However, BASIC doesn't include commands to perform all the detail work. For instance, when you double-click a GEM application from the desktop, you'll often see a rapidly expanding box which grows



to the ultimate size of the application's window. Conversely, when you close an application, a shrinking box appears to indicate graphically that the program has shut down. These effects are created with two matching AES routines known as `graf_growbox` and `graf_shrinkbox`.

### Program 6. Growing and Shrinking Boxes

```
100 REM graf_growbox (AES 73) and graf_shrinkbox
    (AES 74)
110 FULLW 2: CLEARW 2
120 PRINT "Move mouse pointer..."
130 PRINT "Hold button to quit"
140 a#=GB :REM Get the key to Pandora's box
150 gintin=PEEK(a#+8) :REM From me to AES
160 gintout=PEEK(a#+12) :REM From AES to me
170 REM Read pointer position, button
180 POKE CONTRL,124:VDISYS(0)
190 x=PEEK(PTSOUT):y=PEEK(PTSOUT+2)
200 IF PEEK(INTOUT)=1 THEN CLOSEW 2:END
210 REM Set starting box's location/size
220 POKE gintin, 1 :REM x coord
230 POKE gintin+2, 1 :REM y coord
240 POKE gintin+4, 9 :REM width
250 POKE gintin+6, 9 :REM height
260 REM Set ending box's location/size
270 POKE gintin+8, x :REM x coord
280 POKE gintin+10, y :REM y coord
290 POKE gintin+12,50 :REM width
300 POKE gintin+14,50 :REM height
310 GEMSYS(73) :REM Draw growing box
320 GEMSYS(74) :REM Draw shrinking box
330 IF PEEK(gintout)=0 THEN PRINT "Bombed":END
340 GOTO 180
```

When you run Program 6, the growing box moves from the upper left corner of the screen to the current mouse position; the shrinking box moves from the pointer back to the corner again. Move the pointer to several different screen locations to confirm that the boxes can appear anywhere, even on the window borders, without disturbing the underlying graphics.

Since `graf_growbox` and `graf_shrinkbox` are very similar, Program 6 uses the same BASIC sequence to perform the setup for both routines (lines 220–300); then it calls both in succession (lines 310–320). These routines require considerably

more setup than the ones we've seen so far. First, you must tell AES the screen location of the first box in the animated series by supplying its  $x$  (horizontal) and  $y$  (vertical) coordinates. These values are POKEd into locations GINTIN and GINTIN+2, respectively, and indicate the location of the starting box's upper left corner. Next, you must supply the width and height of the starting box in terms of pixels. These values are POKEd into locations GINTIN+4 and GINTIN+6.

After defining the location and size of the starting box, you must define the same values for the ending box in the animated series. POKE the  $x$  and  $y$  coordinates of the ending box's upper left corner into locations GINTIN+8 and GINTIN+10. To indicate the size of the ending box, POKE its width and height (in pixels) into GINTIN+12 and GINTIN+14, respectively.

The actual values that you supply for the starting and ending boxes will depend on whether you're calling `graf_growbox` or `graf_shrinkbox`. When you want the box to grow, define a small starting size and a large ending size; then call `graf_growbox` with GEMSYS(73). To display a shrinking box, you'll need to start with a large box and end with a small one, calling GEMSYS(74).

Many other GEM routines are useful in the BASIC environment. Of course, in order to use any system routine, you must know its opcode as well as the setup procedures it requires. In addition to knowing about specific routines, you will find it helpful to have a general understanding of GEM and how its various components work together. You can find additional information about these subjects in *COMPUTE!'s ST Programmer's Guide*.





## CHAPTER FIVE

---

# Sound and Graphics





# ST Graphics

Brian Flynn

---

*This article delves into the creation of graphics on the ST, showing how to use BASIC statements for drawing lines and shapes and how to add VDI routines to your games and applications software.*

Have you seen the Atari demos of the seagull soaring majestically along the rocky ocean coast and the soccer ball bouncing endlessly back and forth? If you haven't, then quickly beg or borrow copies of the demo disks and run them. You won't be disappointed, for the impressive displays of bird and ball illustrate the tremendous picture-drawing and animation capabilities of the 520 and 1040 computers.

In this article we'll explore how to produce high-quality graphics on the ST color and monochrome monitors. The first section will describe and illustrate each of the BASIC statements for drawing lines, circles, and other shapes. The second section will cover Virtual Device Interface routines. These lightning-fast, high-powered POKE procedures are almost indispensable in many games and in other types of applications software. In the final section, I'll share some tricks and tips on using BASIC and VDI graphics in the same program.

Each of the BASIC programs presented here runs on both the monochrome and color monitors. If you're using the color monitor, however, set the resolution to medium when you first load the *ST Language Disk* (click on the Set Preferences bar in the Options menu).

## Rudiments of ST Graphics

With ST BASIC's powerful and easy-to-use graphics, you can draw anything from boxes and bears to circles and clowns. Indeed, with enough time and patience you could even draw something as complicated as the Atari computer itself.

There are nine statements in the Atari's graphics vocabulary; each is explained in Table 1. You can use these statements to draw shapes, to fill the shapes with different patterns, and to add color to your creations.



**Table 1. Graphics Statements**

In using the following graphics statements, remember that the Atari always measures distances in pixels, that  $x$  values run horizontally and  $y$  values vertically, and that the point (0,0) is the upper left corner of the screen.

**FULLW 2**

Expands the Output window to its maximum size.

**CLEARW 2**

Clears the Output window.

**COLOR** *text color, fill color, line color, index, style*

The first three parameters set colors, and the last two establish the type of pattern used in filling a shape (see Program 2).

**FILL**  $x, y$

Fills a polygon (a closed figure with three or more sides) with the current fill-color. The  $x$  and  $y$  values are the coordinates of any pixel within the boundaries of the shape.

**CIRCLE**  $x, y, \text{radius}, \text{start angle}, \text{end angle}$

The first two parameters are the horizontal and vertical coordinates of the circle's center. The next value is the radius of the circle, or the distance in pixels from the center to the boundary.

The last two parameters are optional. They're used to draw shapes that resemble the slices of a pie. The angles are measured in tenths of degrees (0-3600). Specifying *start* and *end* values of 0 and 900, for example, results in a quarter-circle.

**PCIRCLE**  $x, y, \text{radius}, \text{start angle}, \text{end angle}$

This statement draws a solid rather than a hollow shape. Otherwise, it works just like CIRCLE.

**ELLIPSE**  $x, y, x\text{-radius}, y\text{-radius}, \text{start angle}, \text{end angle}$

An ellipse is an oval, or egg-shaped, figure. The parameters are similar to those of the CIRCLE statement, except that radii are specified for each axis.

**PELLIPSE**  $x, y, x\text{-radius}, y\text{-radius}, \text{start angle}, \text{end angle}$

Draws a solid rather than a hollow ellipse.

**LINEF**  $x1,y1,x2,y2$

Draws a straight line from pixel ( $x1,y1$ ) to pixel ( $x2,y2$ ).

### Shapes

Program 1 draws a box, circle, ellipse, and slice of pie on your screen. The program is written in modular style for ease of understanding, but remember that computer programming isn't a spectator sport. So take two minutes to key in the program (use automatic line numbering from the Command window) and run it.

#### Program 1. Figure Drawing

```
100 REM PROGRAM 1-1
110 GOSUB KEYVALUES
120 GOSUB DRAW.BOX
130 GOSUB DRAW.OTHER.SHAPES
140 GOSUB GOODBYE
150 END
160 '
170 KEYVALUES:
180 DEFINT A-Z
190 FULLW 2: CLEARW 2
200 ST = 3 - PEEK(SYSTAB): REM SCREEN TYPE (1=COLOR; 2=MCHROME)
210 REM X & Y OFFSETS FOR BOX
220 DATA -20,-10, -20,10, 20,10
230 DATA 20,-10, -20,-10
240 FOR I=1TO 5
250 READ X.OFFSET(I),Y.OFFSET(I)
260 NEXT
270 REM RADII
280 RADIUS = 30
290 X.RADIUS = 40
300 Y.RADIUS = 10*ST
310 COLOR 1,1,1,1,1
320 RETURN
330 '
340 DRAW.BOX:
350 X0 = 120: Y0 = 30*ST
360 FOR I=1TO 4
370 X1 = X0+X.OFFSET(I) : Y1 = Y0+Y.OFFSET(I)
380 X2 = X0+X.OFFSET(I+1): Y2 = Y0+Y.OFFSET(I+1)
390 LINEF X1,Y1,X2,Y2
400 NEXT
410 RETURN
420 '
430 DRAW.OTHER.SHAPES:
440 PCIRCLE 240,60*ST,RADIUS
450 PELLIPSE 360,90*ST,X.RADIUS,Y.RADIUS
460 PCIRCLE 480,120*ST,RADIUS,0,900
```



## CHAPTER FIVE

---

```
470 RETURN
480 '
490 GOODBYE:
500 GOTOXY 1,16: PRINT "Press any key"
510 S = INP(2)
520 CLEARW 2
530 RETURN
```

Here's how the program works. First, line 190 expands the Output window to full width (FULLW 2) and clears it (CLEARW 2). You'll probably want to do this in all of your BASIC programs.

Next, line 200 creates a variable called Screen Type, or ST for short. Location SYSTAB (this is a reserved word in BASIC) contains a 1 for the monochrome monitor and a 2 for the color monitor in medium resolution. Hence, 3 minus this value gives a 1 for the color screen (which uses 200 pixels vertically) and a 2 for the black-and-white (which uses 400 pixels vertically). Simply multiplying all *y*-axis shape dimensions by ST, then, makes every figure the same size on both monochrome and color monitors (each uses 600 pixels horizontally). This makes intuitive sense, and I use the variable in almost every program I write.

The DATA statements of lines 220 and 230 contain *x* and *y* offsets from a central point (X0,Y0) in line 350. The first pair of values corresponds to the upper left corner of the box, and the rest continue in counterclockwise fashion. To change the box's size, simply change the offsets.

While the PCIRCLE and PELLIPSE statements are straightforward, it's instructive to change the values of the radii to see the impact. Finally, line 510 keeps the Output window active until an input (INP) is received from the keyboard (device 2).

### Fill

Unlike many other BASIC dialects that I've used, ST BASIC has a variety of handy patterns that you can use to fill shapes. To see what's available, enter and run Program 2.

#### Program 2. Patterns

```
100 REM PROGRAM 1-2
110 GOSUB KEYVALUES
120 GOSUB BOXES
130 GOSUB GOODBYE
```

```
140 END
150 '
160 KEYVALUES:
170 DEFINT A-Z
180 FULLW 2: CLEARW 2
190 ST = 3 - PEEK(SYSTAB): REM SCREEN TYPE (1=CO
    LOR; 2=MCHROME)
200 REM X & Y OFFSETS FOR BOX
210 DATA -10,-10, -10,10, 10,10
220 DATA 10,-10, -10,-10
230 FOR I=1TO 5
240 READ X.OFFSET(I),Y
250 Y.OFFSET(I) = Y*ST
260 NEXT
270 RETURN
280 '
290 BOXES:
300 FOR STYLE = 0 TO 4
310 FOR INDEX = 1 TO 24
320 X0 = INDEX*25-10: Y0 = STYLE*25*ST + 15*ST
330 ' DRAW & FILL BOX
340 FOR I=1TO 4
350 X1 = X0+X.OFFSET(I) : Y1 = Y0+Y.OFFSET(I)
360 X2 = X0+X.OFFSET(I+1): Y2 = Y0+Y.OFFSET(I+1)
370 LINEF X1,Y1,X2,Y2
380 NEXT I
390 COLOR 1,1,1,INDEX,STYLE
400 FILL X0,Y0
410 ' CONTINUE
420 NEXT INDEX,STYLE
430 RETURN
440 '
450 GOODBYE:
460 GOTOXY 1,16: PRINT "Press any key"
470 S = INP(2)
480 CLEARW 2
490 RETURN
```

The Atari draws five rows of 24 boxes. The rows correspond to these five styles:

- 0 Hollow
- 1 Solid
- 2 Patterns
- 3 Hatching
- 4 Atari logo

The columns correspond to an index for each particular style. Index 22 and style 2, for example, together give a checkered pattern.



The correct syntax for filling shapes is

**COLOR 1,1,1,INDEX,STYLE**

You can substitute other numbers for the 1's (see line 390). Experiment with COLOR by changing the 1's to other numbers between 1 and 16.

You might want to save this program on disk. Then you can easily access it to select a fill pattern for any application.

### Color

The Atari has 512 different colors. In medium resolution on the color monitor, when you first boot up your system, four are available at one time—white, black, red, and green. Using these same colors in program after program, however, becomes awfully dull. Fortunately, there's an easy way to access the others from BASIC. Take a look at Program 3.

#### Program 3. Color Changes

```
100 REM PROGRAM 1-3
110 GOSUB KEYVALUES
120 GOSUB CIRCLES
130 GOSUB GOODBYE
140 END
150 '
160 KEYVALUES:
170 DEFINT A-Z
180 RANDOMIZE 0
190 FULLW 2: CLEARW 2
200 ' INITIAL COLORS (WHITE, BLACK, BLACK, BLA
    CK)
210 DATA 1911,0,0,0
220 FOR I=0TO 3: READ KOLOR(I): NEXT
230 ' PUT THEM INTO MEMORY
240 LC# = 1114: POKE LC#,VARPTR(KOLOR(0))
250 RETURN
260 '
270 CIRCLES:
280 ' GET TWO RANDOM COLORS
290 FOR I=1TO 2
300 RED(I) = 8*RND
310 GREEN(I) = 8*RND
320 BLUE(I) = 8*RND
330 KOLOR(I) = 256*RED(I) + 16*GREEN(I) + BLUE(I)
340 NEXT
350 ' PUT THEM INTO MEMORY
```

```

360 POKE LC#,VARPTR(KOLOR(0))
370 ' DRAW CIRCLES
380 FOR I=2 TO 3
390 COLOR 1,I,I,1,1
400 PCIRCLE 200*(I-1),80,50
410 GOTOXY 22*I-24,12: PRINT KOLOR(I-1);SPACE$(3
)
420 NEXT
430 ' CONTINUE
440 GOTOXY 15,15
450 PRINT "Press Q to Quit, or any other key to
continue"
460 X = INP(2)
470 IF X <> 81 AND X <> 113 THEN CIRCLES
480 RETURN
490 '
500 GOODBYE:
510 ' RESET COLORS
520 CLEARW 2
530 KOLOR(1) = 1570: KOLOR(2) = 609
540 POKE LC#,VARPTR(KOLOR(0))
550 RETURN

```

Let's examine Program 3. First, in the KEYVALUES subroutine, four colors are read into a vector called KOLOR. Line 240 POKES into location 1114 (LC#), the starting address of the vector. Note that VARPTR, variable pointer, is a reserved word in BASIC; it tells where a variable is stored. When the magic location 1114 is not a zero, the Atari knows not to rely on default colors, but instead to branch to the address that contains the new palette.

But why is white equal to a weird number like 1911? The Atari produces screen colors based on eight different intensities (0-7) of the primary colors red, green, and blue ( $8 \times 8 \times 8 = 512$ ). Each screen color is stored in a 3-bit field using the 11 rightmost bits of a 16-bit byte:

Red	Green	Blue
<div style="display: flex; align-items: center; justify-content: center;"> <span style="margin-right: 5px;">{</span> <span style="margin-right: 5px;">}</span> </div> 0 0 0 0 0 1 1 1 0	<div style="display: flex; align-items: center; justify-content: center;"> <span style="margin-right: 5px;">{</span> <span style="margin-right: 5px;">}</span> </div> 0 1 1 1 0 1 1 1 0	<div style="display: flex; align-items: center; justify-content: center;"> <span style="margin-right: 5px;">{</span> <span style="margin-right: 5px;">}</span> </div> 0 1 1 1 1 0 1 1 1
$7 \times 256$	$7 \times 16$	$7 \times 1$

Turning on the primary colors to full intensity produces white (11101110111 binary = 1911 decimal). Turning them off produces black.

With this arithmetic in mind, lines 290-340 of the program randomly generate two different screen colors. And lines



380-420 use them to color circles.

There's only one more point to note. The Atari stores colors internally in a different order from the way you access them from BASIC:

Internal Code	BASIC Color Code	Default Color
0	→ 0	white
15	→ 1	black
1	→ 2	red
2	→ 3	green

This is why line 380 reads `I=2 TO 3` rather than `I=1 TO 2`.

Finally, under each circle, the program displays the number of the screen color that's been shown. You may want to jot down the numbers of the colors that you find pleasing for use in your own programs.

## The VDI Connection

For many applications, ST BASIC's graphics statements like `CIRCLE` and `LINEF` are fine. In other cases, however, greater speed and flexibility are needed. That's where the virtual device interface (VDI) routines come in handy. These built-in `POKE` procedures enable you to perform scores of useful operations.

In this section we'll discuss three of the more valuable graphics routines—writing text to any screen location, drawing a box, and drawing any kind of shape at all. These are routines that you'll find yourself using in program after program.

## Writing Text

When you run a BASIC program, the word *OUTPUT* appears at the top of the screen. Wouldn't it be nice to change this title to something more meaningful? Program 4 shows you how.

### Program 4. New Screen Titles

```

1000 REM PROGRAM 1-4
1100 GOSUB KEYVALUES
1200 GOSUB TITLEBAR
1300 GOSUB GOODBYE
1400 END
1500 '
1600 KEYVALUES:
```

```
170 DEFINT A-Z
180 FULLW 2: CLEARW 2
190 ST = 3 - PEEK(SYSTAB): REM SCREEN TYPE (1=CO
    LOR; 2=MCHROME)
200 RETURN
210 '
220 TITLEBAR:
230 T$ = " New Title Bar "
240 AT$ = CHR$(14)+CHR$(15)
250 T$ = CHR$(32) + AT$ + T$ + AT$ + CHR$(32)
260 COLOR 2
270 Y = 15*ST+4: CN$ = "ON": GOSUB TEXT.WRITE
280 RETURN
290 '
300 TEXT.WRITE:
310 LN = LEN(T$)
320 IF CN$ = "ON" THEN X = 312 - LN*8/2
330 POKE CONTRL,8: POKE CONTRL+2,1: POKE CONTRL+
    6,LN
340 FOR Q = 1 TO LN
350 POKE INTIN + (Q-1)*2,ASC(MID$(T$,Q,1))
360 NEXT Q
370 POKE PTSIN,X: POKE PTSIN+2,Y
380 VDISYS(0)
390 RETURN
400 '
410 GOODBYE:
420 COLOR 1
430 GOTOXY 1,16: PRINT "Press any key"
440 S = INP(2)
450 CLEARW 2
460 RETURN
```

First, the TITLEBAR subroutine establishes the new title and embellishes it with the Atari logo (lines 230-250). Next, line 270 computes the y coordinate of the pixel to write to (Y=19 for the color screen and Y=34 for the monochrome). It also sets the variable CN\$ to "ON" for centering the expression horizontally.

The TEXT.WRITE subroutine is the heart of the program, and I use it in many applications. Lines 330-380 are the actual VDI procedure; the following BASIC reserved words are worth noting:

Word	Meaning
CONTRL	Control
INTIN	Information input
PTSIN	Points input



These words represent locations in memory, and storing values in them tells the Atari what to do. POKE CONTRL,8 in line 330, for example, invokes the routine for displaying text. POKE CONTRL+6, LN stores the length of the title. Lines 340-360 place in memory the ASCII code of each character of the string. And line 370 stores the  $x$  and  $y$  coordinates of the pixel to write to. Finally, line 380 executes the VDI routine.

You can use this procedure not only for title bars, but also to write anything anywhere on the screen. Simply place your string in T\$, turn CN\$ on or off as appropriate, assign a pixel coordinate to  $y$  (and to  $x$  if necessary), and then GOSUB TEXT.WRITE.

### Drawing a Box

ST BASIC lacks a command for drawing a rectangle in a single swoop. Instead, you must use the LINEF statement, with all four corners addressed.

Program 5 presents a VDI routine for drawing and filling a box. It's easy to use and extremely fast. Lines 210-230 define the horizontal and vertical dimensions of the shape. Note that YDELTA on the monochrome screen (ST=2) is twice the pixel length of its color-screen counterpart.

### Program 5. Box Fill

```
100 REM PROGRAM 1-5
110 GOSUB KEYVALUES
120 GOSUB PICTURE
130 GOSUB GOODBYE
140 END
150 '
160 KEYVALUES:
170 DEFINIT A-Z
180 FULLW 2: CLEARW 2
190 ST = 3 - PEEK(SYSTAB): REM SCREEN TYPE (1=CO
    LOR; 2=MCHROME)
200 ' X & Y OFFSETS FOR BOX
210 DATA 20,10
220 READ XDELTA, Y
230 YDELTA = Y*ST
240 COLOR 1,2,1,4,2
250 RETURN
260 '
270 PICTURE:
280 Y = 80*ST
```

```
290   FOR I=1 TO 10
300   X = 50*I+25: GOSUB DRAW.BOX
310   NEXT
320   RETURN
330   '
340   DRAW.BOX:
350   POKE CONTRL,11: POKE CONTRL+2,2: POKE CONTRL
      +6,0
360   POKE CONTRL+10,1
370   POKE PTSIN,X-XDELTA : POKE PTSIN+2,Y-YDELTA
380   POKE PTSIN+4,X+XDELTA: POKE PTSIN+6,Y+YDELTA
390   VDISYS(0)
400   RETURN
410   '
420   GOODBYE:
430   GOTOXY 1,16: PRINT "Press any key"
440   S = INP(2)
450   CLEARW 2
460   RETURN
```

The PICTURE subroutine calls for drawing ten boxes. DRAW.BOX (line 340) is the actual VDI procedure. Here's how it works. First, the 11 (line 350) is the operations code (opcode, for short) that denotes the routine for drawing one of ten built-in shapes. Next, the number 2 is POKED into memory. This tells VDI that there are two key vertices to our figure: the upper left corner of the box and the lower right corner. Providing the other two corners, by the way, would give superfluous information since a rectangle is uniquely defined by one corner and its opposite. POKE CONTRL+10,1 tells VDI to draw a bar (8 would produce a rounded rectangle), and the next two lines give the *x* and *y* coordinates of our two corners. Finally, VDISYS(0) executes the routine.

### Drawing a Windmill

Circles, pies, and ellipses, and boxes, bars, and rectangles are nice, but what about drawing something different? As Program 6 shows, this is easy with VDI.



## CHAPTER FIVE

### Program 6. Odd Shapes

```
100 REM PROGRAM 1-6
110 GOSUB KEYVALUES
120 GOSUB MILL.DATA
130 GOSUB VANE.DATA
140 GOSUB DRAW.WINDMILL
150 GOSUB GOODBYE
160 END
170 '
180 KEYVALUES:
190 DEFINT A-Z
200 FULLW 2: CLEARW 2
210 ST = 3 - PEEK(SYSTAB): REM SCREEN TYPE (1=COLOR; 2=MCHROME)
220 DIM X.MILL(12), Y.MILL(12), X.VANE(18), Y.VANE(18)
230 RETURN
240 '
250 MILL.DATA:
260 DATA -2, -2, -12, 7, -12, 11, -3, 11, -3, 6, 4, 6, 4, 11, 13, 11
270 DATA 13, 7, 3, -2, -2, -2
280 FOR I=1 TO 11
290 READ X.MILL(I), Y
300 Y.MILL(I) = ST*Y
310 NEXT
320 RETURN
330 '
340 VANE.DATA:
350 DATA -1, -6, -8, -12, -15, -8, -3, -5, -3, -4, -15, -1
360 DATA -8, 3, -1, -3, 2, -3, 9, 3, 16, -1, 4, -4, 4, -5, 16, -8
370 DATA 9, -12, 2, -6, -1, -6
380 FOR I=1 TO 17
390 READ X.VANE(I), Y
400 Y.VANE(I) = ST*Y
410 NEXT
420 RETURN
430 '
440 DRAW.WINDMILL:
450 X = 310: Y = 80*ST
460 COLOR 1, 2, 1, 1, 1
470 ' MILL
480 POKE CONTRL, 9: POKE CONTRL+2, 11: POKE CONTRL+6, 0
490 CNT = 0
500 FOR I=1 TO 11
510 POKE PTSIN+CNT, X+X.MILL(I)
520 POKE PTSIN+CNT+2, Y+Y.MILL(I)
```

```
530 CNT = CNT + 4
540 NEXT I
550 VDISYS(0)
560 ' VANE
570 COLOR 1,3,1,4,2
580 POKE CONTRL,9: POKE CONTRL+2,17: POKE CONTRL
    +6,0
590 CNT = 0
600 FOR I=1TO 17
610 POKE PTSIN+CNT,X+X.VANE(I)
620 POKE PTSIN+CNT+2,Y+Y.VANE(I)
630 CNT = CNT + 4
640 NEXT I
650 VDISYS(0)
660 RETURN
670 '
680 GOODBYE:
690 GOTOXY 1,16: PRINT "Press any key"
700 S = INP(2)
710 CLEARW 2
720 RETURN
```

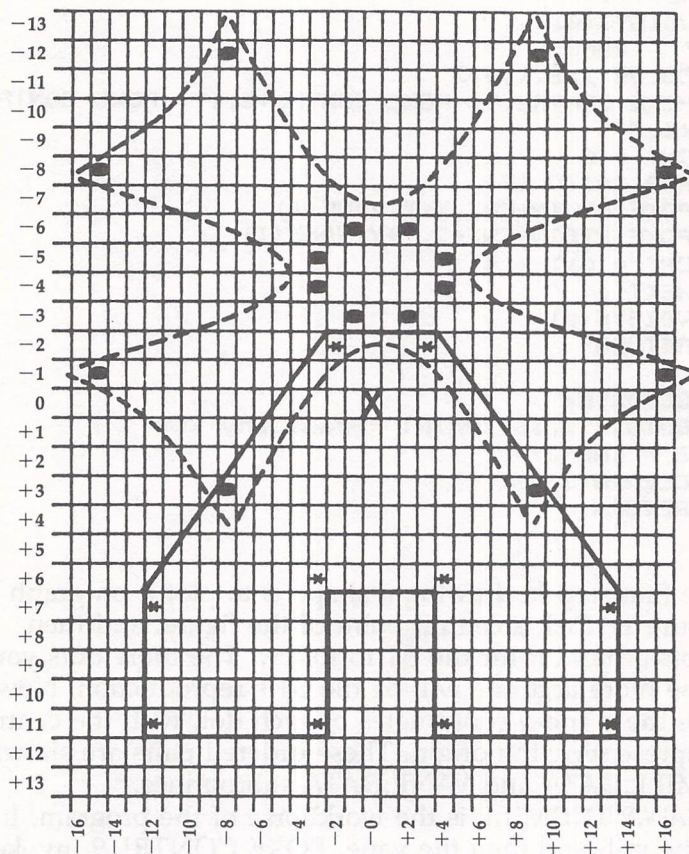
The first step in drawing a shape is to plot it on graph paper (Figure 1). Pick a central point of the figure, and then draw dots around it for the ST to follow. The more dots you draw, the more accurate will be the ST's reproduction. Now, compute the  $x$  and  $y$  coordinates of each dot, with the central point representing the origin. These ordered pairs are shown in the MILL.DATA and VANE.DATA subroutines.

DRAW.WINDMILL is the workhorse of the program. It draws the mill and then the vane. POKE CONTRL,9 invokes the VDI routine for drawing and filling a polygon. The 11 in line 480 represents the number of points in the windmill, with the  $x$  and  $y$  coordinates POKEd into memory in lines 500-540. Drawing the vane works the same way.

I've used this VDI procedure to draw knights, horses, monkeys, and rabbits. The time-consuming part is plotting the points on paper. But it's well worth the effort when you see the ST faithfully reproduce the shapes on your monitor.



Figure 1. Windmill



## Integrating BASIC and VDI

Now, let's draw two boxes on the screen, each the same size and located side by side (Program 7). The only catch is that we'll draw one using BASIC's LINE# statement and the other using VDI.

### Program 7. Drawing Boxes

```

100 REM PROGRAM 1-7
110 GOSUB KEYVALUES
120 GOSUB BASIC.BOX
130 GOSUB VDI.BOX
140 GOSUB GOODBYE
150 END
160 '
    
```

```
170 KEYVALUES:
180 DEFINT A-Z
190 FULLW 2: CLEARW 2
200 ST = 3 - PEEK(SYSTAB): REM SCREEN TYPE (1=COLOR; 2=MCHROME)
210 IF ST = 1 THEN CS = 3 ELSE CS = 0: REM COLOR SCREEN
220 COLOR 1,2,2,1,1
230 RETURN
240 '
250 BASIC.BOX:
260 ' X & Y COORDINATES OF
270 ' UPPER-LEFT & LOWER-RIGHT CORNERS
280 X1=200: Y1=100: X2=250: Y2=150
290 LINEF X1,Y1,X1,Y2: LINEF X1,Y2,X2,Y2
300 LINEF X2,Y2,X2,Y1: LINEF X2,Y1,X1,Y1
310 FILL 225,125
320 RETURN
330 '
340 VDI.BOX:
350 X1=300: X2=350: ' NEW X COORDINATES
360 POKE CONTRL,11: POKE CONTRL+2,2: POKE CONTRL+6,0
370 POKE CONTRL+10,1
380 POKE PTSIN,X1: POKE PTSIN+2,Y1
390 POKE PTSIN+4,X2: POKE PTSIN+6,Y2
400 VDISYS(0)
410 RETURN
420 '
430 GOODBYE:
440 GOTOXY 1,16: PRINT "Press any key"
450 S = INP(2)
460 CLEARW 2
470 RETURN
```

The first box goes from coordinates (200,100) in its upper left corner to coordinates (250,150) in its lower right. The second goes from (300,100) to (350,150). In both cases, then, the y coordinates are the same (100 and 150). Hence, the boxes should appear at the same level, right?

When Program 7 runs, however, we find that VDI draws its box higher. This is because to VDI row 0 is the top edge of the screen (Table 2). But to BASIC it's the first line beneath the title bar.



**Table 2. BASIC vs. VDI Vertical Positions**

Color Screen			VDI Line	
Title Bar				21
ST BASIC Line			0	22
			1	23
			2	24
Monochrome Screen			VDI Line	
Title Bar				37
ST BASIC Line			0	38
			1	39
			2	40

You'll need to keep this anomaly in mind whenever you use VDI and BASIC graphics in the same program.

Fortunately, this quirk is not very difficult to overcome. I use the magic formula  $19*ST+CS$ . The variable ST is familiar from our earlier programs; it stands for Screen Type (1 = medium-resolution color; 2 = monochrome). The variable CS equals 3 for the Color Screen and 0 for the black-and-white. Hence, the formula equals either 22 or 38, values that jibe perfectly with the rows in Table 2.

To fix the glitch with our boxes, then, adjust the *y* coordinates by adding these lines to the program.

```
355 Y1=Y1+19*ST+CS
```

```
356 Y2=Y2+19*ST+CS
```

The BASIC and VDI graphics routines presented here will enable you to draw a wide variety of fancy shapes and figures. You may want to use ST graphics to enhance a game or to embellish a financial, home, or statistical application.

For additional VDI routines, *COMPUTE!'s ST Programmer's Guide* is a source that I've found useful and understandable. In any event, there's enough in this article to get you started—and who knows? You may end up drawing a soaring gull or a bouncing ball.

# MODified Shapes for Atari ST

Robert G. Geiger

---

*With this program, you can create pleasing graphics and also gain valuable information about using GEMSYS and VDISYS in ST BASIC. With the techniques explained here, you can draw on a full-screen graphics area (without BASIC's usual window borders), manipulate dialog boxes, and monitor mouse events.*

Paul Carlson's article "MODified Shapes for IBM" (*COMPUTE!* magazine, May 1986) is interesting both as a tutorial on the MOD operator and for its outstanding graphics. Since ST BASIC also has the MOD operator, the logic used in the IBM program works equally well on the Atari ST. But the ST is capable of doing much more. With the aid of GEMSYS and VDISYS, not only can you replicate the original program, but you can also add such distinctive ST features as dialog boxes and mouse input.

Type in "MODified Shapes for ST" and save a copy before you run it. When you type the program, you'll notice that several lines (those containing VDISYS or GEMSYS calls) are more than 80 characters long. This is done so that all the information for each GEM call is on one program line. The ST BASIC editor lets you enter lines up to 255 characters in length if the first character in the second screen line is a space.

If you have a 520ST with 512K RAM and the TOS operating system on disk instead of in ROM (Read Only Memory), you must turn off buffered graphics before you run the program. If your ST has more than 512K of memory or TOS in ROM, you should have enough memory to run the program without taking this step.

The program runs in any screen resolution—low or medium resolution on a color monitor, or high resolution on a monochrome monitor. However, low resolution is truest to the four-color IBM screen used in the original program. In medium or high resolution, the design occupies only part of the screen.



### From PC to ST

If you're familiar with IBM BASIC, you may find it instructive to compare the original program with the ST version. Some statements in the PC/PCjr program, such as KEY OFF, are unnecessary in ST BASIC and can be omitted. Most of the program logic, which simply manipulates variables, works on the ST with no modification at all.

However, other operations require different commands. For instance, at the conclusion of the IBM program, the INKEY\$ statement is used to make the program pause until you press a key. ST BASIC lacks INKEY\$, but you can substitute the INP(2) function. And though the LINEF command in ST BASIC differs a bit in syntax, it can draw lines much as the IBM version does. The IBM clears the screen with CLS, but the ST uses CLEARW 2, and so on.

It's possible to translate most of the IBM program by making BASIC substitutions, but if you confine yourself to ordinary BASIC commands, you'll end up with a translation that's almost, but not quite, satisfactory. One major problem involves the ST BASIC Output window. When you open the window to full-screen size with FULLW 2: CLEARW 2, part of the visible screen area is taken up by the window border, title line, and menu bar. In low resolution, the usable screen area is fewer than 40 characters wide, and you can print only 17 lines of text before the window's contents begin to scroll upward.

Because screen space is taken up by the window borders, it seems impossible to duplicate the IBM's 320 × 200 pixel screen exactly. Even worse, while IBM BASIC defines the upper left corner of the screen as coordinate (0,0), ST BASIC considers coordinate (0,0) to be the upper left point inside the Output window. As a result, any graphics designed to occupy the entire IBM screen will be clipped in the ST BASIC Output window.

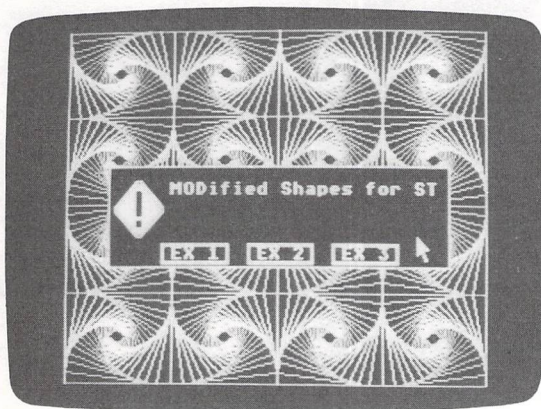
### Full Screens in ST BASIC

The solution is to use system calls for screen output. GEM (Graphics Environment Manager) allows you to draw anywhere on the screen, including the areas normally occupied by the BASIC windows themselves. Two of the more important parts of GEM are the VDI (Virtual Device Interface), which handles low-level mouse input and graphics display, and the

AES (Application Environment Services), which handles more complex routines such as managing windows, drop-down menus, icons, and dialog boxes.

The basic method of calling a VDI routine is to store the information it requires into reserved memory locations which are defined by the reserved variables `CONTRL`, `PTSIN`, and `INTIN`. These memory locations are known as *parameter blocks*. Every VDI routine requires different information, and some VDI routines don't need information in all three parameter blocks. Once this preliminary work is done, you call the VDI routine with the statement `VDISYS(0)`. The zero is a dummy parameter, which can be any numeric value. If you'd like to learn more about `VDISYS` routines, see "Adding System Power to ST BASIC," elsewhere in this book.

The procedure for calling an AES routine is similar. First, you store the information it requires in memory; then you call the routine with a `GEMSYS` statement. But different information must be passed to the routine, and the number enclosed in the parentheses is significant. For instance, `GEMSYS(52)` calls AES routine 52 (see below). This program uses `VDISYS` to create graphics and `GEMSYS` to handle user input.



*"MODified Shapes for Atari ST" demonstrates how to draw graphics on the entire screen surface, including areas normally occupied by BASIC's window borders.*

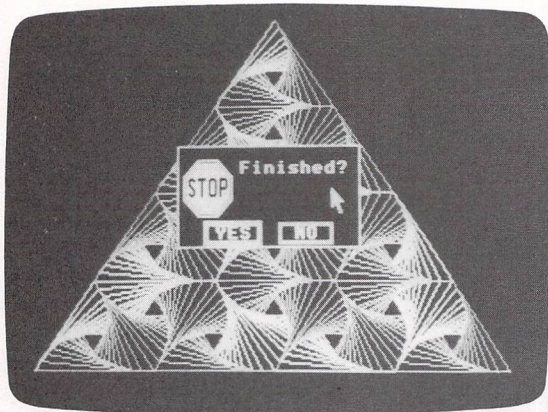
### Dialog Boxes

Some of the most useful AES functions involve various forms of the *dialog box*, a box that appears on top of the current



screen display whenever it's time for you to select an option, respond with a yes or no answer, and so forth. When the interaction is over, GEM restores the screen and lets you continue where you left off. Dialog boxes are a powerful way of creating a friendly atmosphere in your programs. The full capabilities of the dialog box are beyond the scope of BASIC (unless you have the *Resource Construction Set* utility from the *ST Development System*), but two forms of the dialog box—the *alert box* and the *error box*—are available.

When you run MODified Shapes, it begins by displaying a dialog box with three options labeled EX1, EX2, and EX3. Depending on which option you click on, the program will create example screen 1, 2, or 3. When you make a choice, the box disappears, the screen is redrawn, and the program proceeds. This dialog box is created with AES routine 52, known as *FORM\_ALERT*. It creates a dialog box and tells GEM to get input from it, as well. To use *FORM\_ALERT*, you must store two items of information in memory and then call the routine with *GEMSYS(52)*. After the interaction is finished, *FORM\_ALERT* passes one item of information back to you.



*With the aid of GEMSYS, you can call system routines from BASIC to create dialog boxes like the one shown here.*

Most of the information needed by *FORM\_ALERT* can be passed in the form of a BASIC string. First, the string is defined. Then you *POKE* the address of the beginning of the string in a reserved variable area known as *ADDRIN* (*ADDRESS IN*). This tells GEM where the string is located.

The `FORM_ALERT` string begins with a code number indicating which sort of icon you want the box to contain. You may choose a stop sign icon, an exclamation point, or a question mark. These icons appear frequently during GEM desktop operations and are familiar to every ST user. After the icon number comes the text which you want to print inside the box. If an icon is also used, the box has enough room for up to five lines of text.

### Buttons in a Box

The next portion of the string contains the text you want to appear inside the *buttons*. Don't confuse this sort of button with the physical button on the ST mouse device. In this context, a button is a smaller boxed-in area within the dialog box. Point to the dialog button with the mouse and then click the left mouse button to select that option.

You may include as many as three dialog buttons in a single dialog box. If you include just one button, its box may contain up to 20 characters of text. You can also outline one of the buttons with a heavier line to indicate that it can be selected by pressing Return as well as by clicking with the mouse.

Line 70 of the program creates a typical `FORM_ALERT` string. Notice that each component of the string is enclosed in a set of square brackets in this sequence:

**[icon code] [message text] [button text]**

Notice also that new lines within the message text and button text are separated by the logical OR character (`|`). You obtain this character by pressing the backslash key (`\`) while holding down the Shift key.

After creating a string and POKEing its location into memory, you must POKE a value into the location defined as `GINTIN` to indicate which button is to be chosen by pressing Return. POKE 0 into this location to indicate that Return should be ignored. POKE `GINTIN` with 1, 2, or 3 to indicate the first, second, or third button, respectively.

When the `FORM_ALERT` dialog is over, you need some way to learn what choice has been made. This output is returned in the location defined as `GINTOUT`, which you can PEEK from BASIC. When `GINTOUT` equals 1, the first dialog button has been clicked. Values of 2 and 3 indicate that the



second and third dialog buttons have been clicked. Again, keep in mind that these are buttons within the dialog box on the screen, not physical buttons on the mouse.

### Reading Mouse Events

MODified Shapes uses another AES routine, number 21, known as *MOUSE\_EVENT*, to pause until you press both mouse buttons. The *MOUSE\_EVENT* routine requires three inputs, which are passed in locations beginning at GINTIN. The first value to be passed indicates the number of clicks to be detected. The second value indicates the mouse button to be read. And the third indicates the button condition you wish to look for. The number of clicks should be either one or two. For the second value, use the value 1 to indicate the left button, 2 to indicate the right button, and 3 to indicate both buttons. The third value determines which condition—being pressed or not being pressed—the routine checks for. In most cases, this value will be 1, which means that you want to know when the indicated button is pressed. If you supply 0, the routine tells you whether the button is not pressed.

By calling GEM and AES routines, not only can you mimic the IBM's graphics, but you can also add the ST's own signature to the program in the form of dialog boxes and mouse input. The accompanying table summarizes the various VDI and AES routines used in this program, along with the program lines in which each routine is called.

### VDI and AES Routines

#### **Set\_Color Representation**

(Lines 20, 30, 570, 580)

#### **Input Parameters**

POKE CONTRL,14	Opcode
POKE CONTRL+2,0	Number of vertices
POKE CONTRL+6,4	Number of attributes
POKE INTIN,0-15	Number of pen color
POKE INTIN+2,0-1000	Red intensity
POKE INTIN+4,0-1000	Green intensity
POKE INTIN+6,0-1000	Blue intensity

#### **Clear\_Workstation**

(Lines 40, 170, 310, 440)



### **Input Parameters**

POKE CONTRL,3	Opcode
POKE CONTRL+2,0	Number of vertices
POKE CONTRL+6,0	Number of attributes

### **Show\_Cursor**

(Lines 50 and 110)

### **Input Parameters**

POKE CONTRL,122	Opcode
POKE CONTRL+2,0	Number of vertices
POKE CONTRL+6,1	Number of attributes
POKE INTIN,0	Reset flag

Note: The VDI normally makes note internally of how often the HIDE CURSOR call is used. To disable this function, set the reset flag to zero.

### **Form\_Alert**

(Lines 60-80, 120-140)

### **Input Parameters**

POKE GINTIN,0	Button simulated by pressing Return
X#=ADDRIN	ADDRIN is addressed as a double-precision variable
POKE X#,VARPTR(Message\$)	

### **Output Parameters**

KEY=PEEK(GINTOUT)	Value of the button clicked
-------------------	-----------------------------

### **Input Parameters**

POKE CONTRL,123	Opcode
POKE CONTRL+2,0	Number of vertices
POKE CONTRL+6,0	Number of attributes

### **Polyline**

(Lines 240, 380, 510)

### **Input Parameters**

POKE CONTRL,6	Opcode
POKE CONTRL+2,2	Number of vertices
POKE CONTRL+6,0	Number of attributes
POKE PTSIN,X1	x coordinate of first point
POKE PTSIN+2,Y1	y coordinate of first point
POKE PTSIN+4,X2	x coordinate of second point
POKE PTSIN+6,Y2	y coordinate of second point

### **Evnt\_Button**

(Lines 290, 420, 560)

### **Input Parameters**

POKE GINTIN,1-2	Number of clicks for action
POKE GINTIN+2,1-3	Mouse button(s) to be read
POKE GINTIN+4,1	Button condition to detect



## MODified Shapes for ST

```

10  A#=GB:CONTROL=PEEK(A#):GLOBAL=PEEK(A#+4):GIN
    TIN=PEEK(A#+8):GINTOUT=PEEK(A#+12):ADDRIN=PE
    EK(A#+16)
20  POKE CONTRL,14:POKE CONTRL+2,0:POKE CONTRL+6
    ,4:POKE INTIN,0:POKE INTIN+2,0:POKE INTIN+4,
    0:POKE INTIN+6,0:VDISYS(0)
30  POKE CONTRL,14:POKE CONTRL+2,0:POKE CONTRL+6
    ,4:POKE INTIN,1:POKE INTIN+2,1000:POKE INTIN
    +4,1000:POKE INTIN+6,1000:VDISYS(0)
40  POKE CONTRL,3:POKE CONTRL+2,0:POKE CONTRL+6,
    0:VDISYS(0)
50  MAINMENU: POKE CONTRL,122:POKE CONTRL+2,0:PO
    KE CONTRL+6,1:POKE INTIN,0:VDISYS(0)
60  N#=ADDRIN:POKE GINTIN,0:'FORM_ALERT
70  MENU$="[1][Modified Shapes for ST][EX 1]EX
    2]EX 3]" +CHR$(0)+CHR$(0)
80  POKE N#,VARPTR(MENU$):GEMSYS(52)
90  C=PEEK(GINTOUT):POKE CONTRL,123:POKE CONTRL+
    2,0:POKE CONTRL+6,0:VDISYS(0)
100 IF C=3 THEN GOTO EX3 ELSE IF C=2 THEN GOTO E
    X2 ELSE GOTO EX1
110 EXITBOX: POKE CONTRL,122:POKE CONTRL+2,0:POK
    E CONTRL+6,1:POKE INTIN,0:VDISYS(0)
120 M#=ADDRIN:POKE GINTIN,1:'FORM_ALERT box
130 TEXT$="[3][Finished?][YES|NO]" +CHR$(0)+CHR$
    (0)
140 POKE M#,VARPTR(TEXT$):GEMSYS(52):C=PEEK(GINT
    OUT)
150 IF C=2 THEN GOTO MAINMENU ELSE GOTO BYE
160 EX1: SU=.1:RU=1-SU:II=1:C=1
170 POKE CONTRL,3:POKE CONTRL+2,0:POKE CONTRL+6,
    0:VDISYS(0)
180 FOR J=0 to 3:II=-II:JJ=1:FOR I=0 to 6:JJ=-JJ
    :IF I<J or I>6-J THEN 280
190 IF J<2 or I>2 THEN C=C MOD 3+1
200 IF J=3 THEN C=C MOD 3+1
210 X(1)=0:X(2)=39:X(3)=78:Y(1)=0:Y(3)=0:IF II=J
    J THEN Y(2)=48 ELSE Y(2)=-48
220 FOR N=1 to 11:X1=3+X(3)+I*39:Y1=175-Y(3)-J*4
    8+II*JJ*24
230 FOR M=1 to 3:X2=3+X(M)+I*39:Y2=175-Y(M)-J*48
    +II*JJ*24:C=C MOD 3+1
240 COLOR 1,1,C:POKE CONTRL,6:POKE CONTRL+2,2:PO
    KE CONTRL+6,0:POKE PTSIN,X1:POKE PTSIN+2,Y1:
    POKE PTSIN+4,X2:POKE PTSIN+6,Y2:VDISYS(0)
250 X1=X2:Y1=Y2:NJ=M MOD 3+1
260 XD(M)=RU*X(M)+SU*X(NJ):YD(M)=RU*Y(M)+SU*Y(NJ
    ):NEXT M

```

```

270 FOR P=1 to 3:X(P)=XD(P):Y(P)=YD(P):NEXT P,N
280 NEXT I,J
290 POKE GINTIN,1:POKE GINTIN+2,1:POKE GINTIN+4,
    1:GEMSYS(21):GOTO EXITBOX
300 EX2: SU=.12:RU=1-SU
310 POKE CONTRL,3:POKE CONTRL+2,0:POKE CONTRL+6,
    0:VDISYS(0)
320 FOR I=0 to 3:FOR J=0 to 3:IF I MOD 2=J MOD 2
    THEN 340
330 Y(1)=49:Y(2)=0:Y(3)=0:Y(4)=49:GOTO 350
340 Y(1)=0:Y(2)=49:Y(3)=49:Y(4)=0
350 X(1)=20:X(2)=20:X(3)=89:X(4)=89
360 FOR N=0 to 18:X1=X(4)+I*69:Y1=Y(4)+J*49
370 FOR M=1 to 4:X2=X(M)+I*69:Y2=Y(M)+J*49
380 COLOR 1,0,M MOD 2+1:POKE CONTRL,6:POKE CONTR
    L+2,2:POKE CONTRL+6,0:POKE PTSIN,X1:POKE PTS
    IN+2,Y1:POKE PTSIN+4,X2:POKE PTSIN+6,Y2:VDIS
    YS(0)
390 X1=X2:Y1=Y2:NJ=M MOD 4+1
400 XD(M)=RU*X(M)+SU*X(NJ):YD(M)=RU*Y(M)+SU*Y(NJ
    ):NEXT M
410 FOR P=1 to 8:X(P)=XD(P):Y(P)=YD(P):NEXT P,N,
    J,I
420 POKE GINTIN,1:POKE GINTIN+2,1:POKE GINTIN+4,
    1:GEMSYS(21):GOTO EXITBOX
430 EX3: SU=.2:RU=1-SU
440 POKE CONTRL,3:POKE CONTRL+2,0:POKE CONTRL+6,
    0:VDISYS(0)
450 FOR J=0 to 2:FOR I=0 to 2:IF J=0 AND I<>1 TH
    EN 550
460 IF I=1 THEN E=31 ELSE E=0
470 X(1)=0:X(2)=25:X(3)=75:X(4)=100:X(5)=75:X(6)
    =25
480 Y(1)=31:Y(2)=0:Y(3)=0:Y(4)=31:Y(5)=62:Y(6)=6
    2
490 FOR N=0 to 20:X1=35+X(6)+I*75:Y1=223-Y(6)-J*
    62-E
500 FOR M=1 to 6:X2=35+X(M)+I*75:Y2=223-Y(M)-J*6
    2-E
510 COLOR 1,0,M MOD 3+1:POKE CONTRL,6:POKE CONTR
    L+2,2:POKE CONTRL+6,0:POKE PTSIN,X1:POKE PTS
    IN+2,Y1:POKE PTSIN+4,X2:POKE PTSIN+6,Y2:VDIS
    YS(0)
520 X1=X2:Y1=Y2:NJ=M MOD 6+1
530 XD(M)=RU*X(M)+SU*X(NJ):YD(M)=RU*Y(M)+SU*Y(NJ
    ):NEXT M
540 FOR P=1 to 6:X(P)=XD(P):Y(P)=YD(P):NEXT P,N
550 NEXT I,J
560 POKE GINTIN,1:POKE GINTIN+2,1:POKE GINTIN+4,
    1:GEMSYS(21):GOTO EXITBOX

```



## CHAPTER FIVE

```
570 BYE: POKE CONTRL,14:POKE CONTRL+2,0:POKE CONTRL+6,4:POKE INTIN,0:POKE INTIN+2,1000:POKE INTIN+4,1000:POKE INTIN+6,1000:VDISYS(0)
580 POKE CONTRL,14:POKE CONTRL+2,0:POKE CONTRL+6,4:POKE INTIN,1:POKE INTIN+2,0:POKE INTIN+4,0:POKE INTIN+6,0:VDISYS(0):END
```

# NEOchrome: The Rainbow Machine

## Advanced Color Features of NEOchrome

Lee Noel, Jr., and Selby Bateman

---

*Many of the most attractive and powerful features of the NEOchrome painting program from Atari are largely undocumented. Here, an experienced artist shows you how to open up special features of this remarkable graphics program and even produce color animation on your ST. Operates only in low-resolution graphics mode.*

The *NEOchrome Sampler* painting program was shipped free with every ST during the first year of the computer's availability. A full-featured version of *NEOchrome* is now being sold commercially. Both versions offer ST owners excellent graphics software that really takes advantage of the computer's superb color graphics system. Even the *Sampler* (*NEOchrome* version 0.5), while missing a few of the more sophisticated drawing tools of its successor, is an attractive and powerful computer painting package.

In fact, it's difficult to find a graphics program that matches *NEOchrome's* speed, flexibility, and ease of use. Its simplicity also makes it an excellent introduction to the icon-based software environment that has become so popular.

If you look just below the surface, you'll discover *NEOchrome's* special power to create magic with color. This is of little use to owners of monochrome ST systems, but it opens up a world of amazing possibilities for those with color displays. With a basic understanding of these features, you'll be surprised at how easily you can create paintings and designs that show off all of the ST's graphics capabilities. Before exploring some of these features, let's take a quick look at how the ST and *NEOchrome* work with color.



### Using Color

*NEOchrome* operates only in the ST's low-resolution graphics mode. Here, the computer offers a rich palette of 512 different colors, and any 16 of these can be displayed simultaneously.

Most low-resolution graphics programs allow you to set these 16 colors by adjusting their individual RGB levels. (RGB simply stands for red, green, and blue, which are the three primary colors of video display. The different colors you see on your monitor are generated by mixing the three primaries together in various proportions.) In this mode, the ST offers eight levels for each primary color. This gives  $8 \times 8 \times 8$  possible RGB combinations—hence, the ST's 512 colors.

The RGB system may seem complicated at first, but it's really rather straightforward once you examine it. However, the unexpected results of color mixing can frequently baffle even experienced users. For example, yellow is produced by mixing the highest levels of red and green. An artist combining similarly colored paints might expect to see brown as the end product. But the colors we see that come from paint and other pigments are the result of reflected light. Computer-generated colors are directly transmitted light. It's as if the display screen is composed of thousands of tiny, brightly shining light bulbs. These small light bulbs are picture elements, called *pixels* for short, that make up what you see on the screen.

Typically, computer graphics programs allow you to adjust RGB levels with a display made of movable *sliders*, much like those on the control boards found in recording studios. However, without considerable experience, it's hard to predict the exact color results of moving the sliders about. In cases where small adjustments are made, it's almost impossible. The particular mossy green you're trying to create might require the most outlandish RGB values you can imagine. And that can lead to frustration no matter how patient the computer artist.

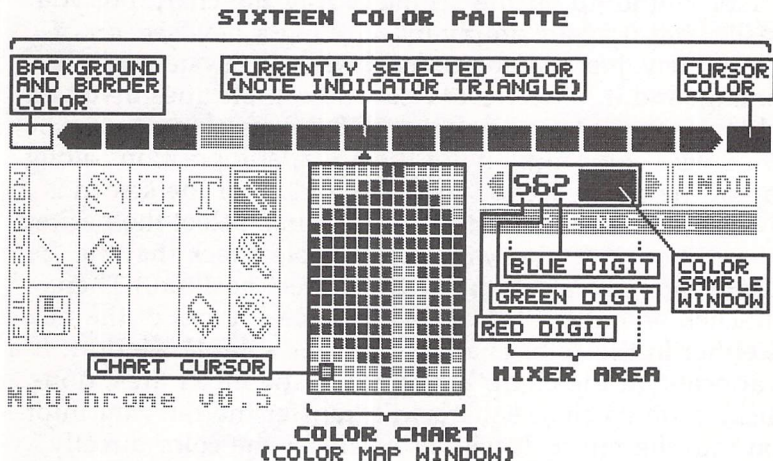
But not if you're using *NEOchrome*. The program provides you with a color chart just like those used by professional graphic designers. Instead of trying to calculate the correct RGB settings, you simply look on the chart for the color you want. Given the ST's polychromatic range, chances are good that you'll find exactly what you're looking for.

### NEOchrome's Color Chart

The chart in the center of the *NEOchrome* menu can display at one time 208 of the ST's 512 colors. All 512 colors are available by horizontally scrolling the chart. You can scroll by holding down the right mouse button and dragging the chart left or right. Since nearly half the ST's possible colors appear together in the chart window, you can also make selections based on nearby alternatives and their appearance relative to the other colors visible in the chart. With this system, you're no longer restricted to blindly juggling RGB numbers. Where color selection is concerned, there's no substitute for the good, old-fashioned human eye.

Transferring colors from the chart into the 16 palette positions is a straightforward operation. Figure 1 will help identify and locate important color chart features.

Figure 1. Color Chart and Palette Features



Select any particular color from the chart by left-clicking on it with the mouse. (Right-clicking won't work. It's reserved for the right-dragging process used to scroll the color chart.) A square cursor indicates the selection, and the appropriate color information is transferred to the mixer area. Here, the color sample window shows a large patch of the selected color, and its RGB values are also displayed.

Although using the numeric RGB information is often



cumbersome when you're initially trying to select colors, having it supplied to you in this way is quite valuable. Keeping a record of RGB values may help you remember a particularly striking color or sequence of colors. Moreover, RGB numbers provide a precise method of describing colors to other ST users. As a result, simple black-and-white printed material can contain exact instructions for coloration of graphic design work.

Once specified, RGB numbers can then be used to select colors directly from the mixer area. The three digits in the mixer correspond in position to the three letters making up the abbreviation RGB. In other words, the leftmost digit is the red value, and the other two complete the pattern. Left-clicking on one of the digits increases its value by one. Right-clicking decreases by the same amount. Changes made to the RGB levels are instantly reflected in the color sample window, which displays the new color, and in the chart itself, where the cursor moves to surround the newly specified color. In many cases, the cursor will jump off the visible part of the chart, but you can easily find it again simply by scrolling a new section of chart into view. Just playing around with this system can be fascinating, and it helps make clear the complexities involved in picking a particular color from RGB values alone.

Moving a color into one of the 16 palette positions along the top of the menu section near the middle of the screen is as simple as a double-click of the mouse. First, select the desired palette position. (It's always best to double-check that the small indicator triangle is below the correct position.) Then double-click on either of the two available sources of the new color, either in the color chart or from the sample window. If a color appears on the chart, but is not in the mixer area, double-clicking on its chart square will transfer the relevant information into the mixer. It will also transfer the color directly into the currently selected palette position. The reverse is also true. Double-clicking on a palette position will transfer that color's RGB values into the mixer area and sample window. The cursor in the color chart will also shift to reflect that color. This technique provides a handy way to check on the RGB values of an established palette. Such values can then be re-recorded prior to any experimental changes to the palette.

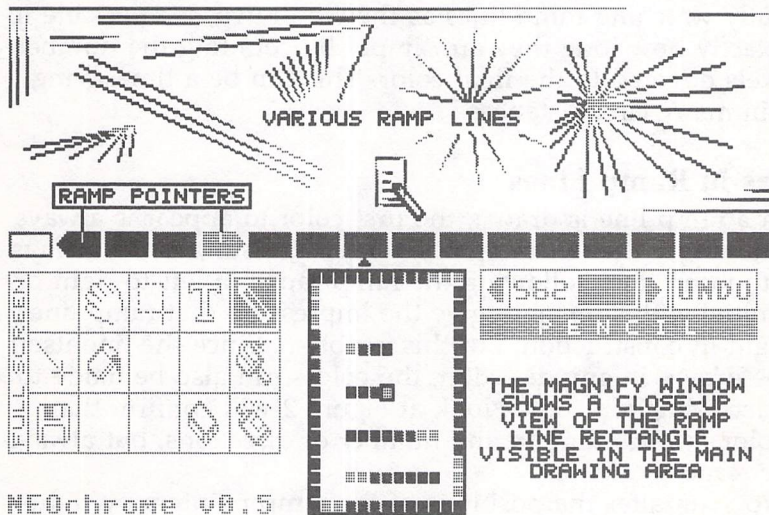
*NEOchrome's* color chart offers an unbeatable combination of simplicity (what you see is what you get) and flexibility (RGB values can be utilized, but without the usual frustration and confusion).

### Ramp Lines

Easy access to the ST's multitude of colors is particularly helpful when you're using another of *NEOchrome's* unusual features, *ramp lines*. Not surprisingly, ramp lines are available only by using the line-drawing tool. The line-drawing tool is as easy to use as *NEOchrome's* other features. Pressing the left mouse button establishes the starting point for a line. Dragging stretches an elastic straight line out to any position indicated by the onscreen mouse pointer, and releasing the left button fixes both the line and its endpoint.

Ramp lines are created with the line-drawing tool in almost exactly the same way as normal lines, but the right mouse button is used instead of the left. Figure 2 shows some important points and gives an idea of the appearance of the ramp lines themselves. Naturally, printing these lines in black and white restricts the illustrations to four tones of gray plus white rather than the colorful combination you'll see on your screen.

Figure 2. Color Ramp Pointers and Ramp Lines





The keys to understanding ramp lines are the ramp pointers, the two pointed palette blocks that you normally see at either end of the *NEOchrome* palette. (These are sometimes referred to as *notches*.) A ramp line follows the path an ordinary line would, but it is made up of all the colors between (including those under) the two ramp pointers.

In Figure 2, there are four colors between the pointers, so all lines made by right-dragging are drawn with these colors. Normally, the small triangular indicator under the palette will dictate the drawing color, but ramp lines always take their colors from the range specified by the pointers.

The magnify window (created in the color chart window whenever the onscreen pointer is in the drawing area) shows details of the short ramp lines drawn in the box above it. The four-pixel-long lines display one pixel of each of the ramp colors. The eight-pixel lines show paired colors. The program routine within *NEOchrome* that forms ramp lines tries to divide the colors evenly along the length of each line, although exact division isn't possible if the length of the line isn't a precise multiple of the number of colors in the ramp range. This unevenness can be seen on the side walls of the magnify box. By carefully watching ramp lines as they're drawn, it's possible to tell exactly how long they are simply by counting the number of pixels devoted to the final color. This can be a timesaving trick in many circumstances.

### Colors in Ramp Lines

When a ramp line is drawn, the first color to appear is always the one under the left ramp pointer. As a result, colors always go into ramp lines following the ramp range in left-to-right order. Initially, this tends to give the impression that ramp lines are rigid in construction, but this is not so. Since the line itself can be drawn in any direction, the colors can also be made to flow accordingly. A quick look at Figure 2 will confirm that the colors are running in any number of directions, but always in the same sequence.

You can alter the positions of the ramp pointers by right-dragging them with the mouse to the desired location. Any block on the palette row is acceptable, and both pointers can even be set in the same position, forming a double-ended block. The sole limitation is that the pointers can never cross over each other.

With these movable pointers, it's easy to establish many different ramp ranges within the full palette. Ramp lines can be drawn with any of them, so legions of differently colored ramp lines can be placed on the screen at the same time. Fortunately, ramp lines always retain the characteristics of the color range with which they were originally drawn. This means that their colors are stable, unless of course there are alterations to the palette positions you've selected for that screen.

### Using Ramp Lines

Although ramp lines are quite beautiful in themselves, it might seem that they have little practical application in the creation of paintings and designs. But, like many first impressions of *NEOchrome*, this one vanishes after a little exploration.

**Figure 3. Ramp Lines Expanded into Shaded Areas**

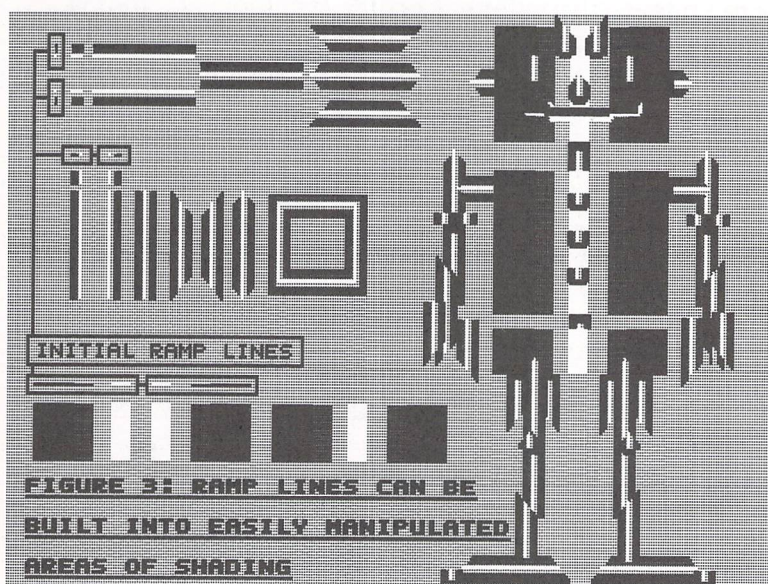


Figure 3 shows how these lines can be applied in a practical manner. Merely by using the copy-box tool, a few short segments of ramp lines can be rapidly expanded into large areas of shade (and color on a monitor screen). As you can see, the original line segments were tiny, but the overall robot assemblage is quite a solid figure. Doing such work is quick



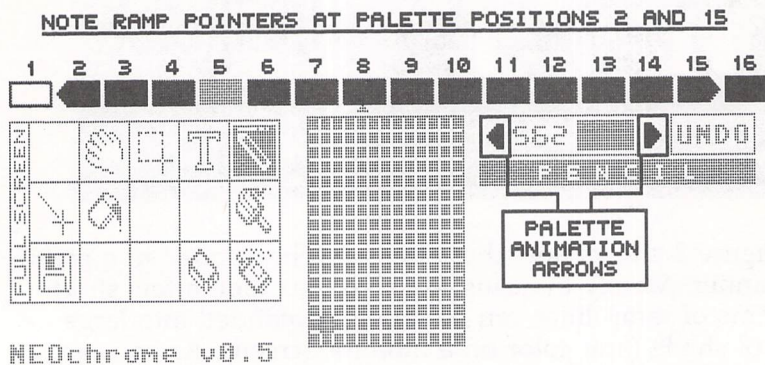
and takes little practice. By preassembling elements like those on the left side of the figure and saving them to disk, it's possible to build up files of ramp line building components for later projects. With an unrestricted palette (again, ours is reduced for printing purposes), such work can be rich, smooth, and detailed. One final point: The barlike objects in the upper left portion of Figure 3 are made of ramp lines which are five pixels long—one pixel for each of the four gray tones, plus white. In most cases, ramp lines look very smooth when they're made of single-pixel color elements. Since such lines are naturally of the minimum possible length, they offer the *NEOchrome* artist a method of producing top-quality graphics with economical use of color.

## Animation

Probably the most impressive feature of *NEOchrome* lies in its ability to animate the palette by color cycling. This, too, is closely tied to the ramp pointers, so ramp lines and palette animation are relatives of a sort. Keeping this in mind may help explain the concept.

As a matter of fact, one of the easiest ways to see color cycling in action is to draw a number of ramp lines on the screen. Then, all you need to do is to right-click the mouse on either of the palette animation arrows that appear on the ends of the mixer area in the *NEOchrome* menu. Figure 4 shows details of the minimal controls needed for cycling effects.

**Figure 4. Animation Features**



In the case of the figure, as soon as one of the animation arrows is activated by a right click, the colors between the ramp pointers at positions 2 and 15 will begin to cycle repeatedly through the entire range. For example, right-clicking on the left animation arrow will initiate a sequence in which the color in position 2 moves into 15, 15 moves into 14, and all the rest shift leftward one place accordingly. Incidentally, left-clicking on either arrow will shift the ramp range one step in the direction indicated by the arrow, but not continuously. This effect can be useful for making adjustments to a static palette or for a step-by-step preview of full-scale color cycling.

### Operating Color Cycling

Once color cycling has been set up, ramp lines drawn with the specified range take on a life of their own. Depending on the direction in which a line has originally been drawn, the ramp colors will appear to move through it. Cycling started by right-clicking on the other palette animation arrow will, of course, reverse the flow of color 180 degrees.

Other rules for operating the system are equally straightforward. Once cycling has started, you can stop it by a left or right click on the *opposite* arrow. Restarting animation then takes another right click. If cycling has been established, its rate can be increased by left-clicking on the active arrow. The rate is decreased by right-clicking.

*NEOchrome* can also remember animation information. If you establish an animation sequence and save the picture to disk *while it's still running*, speed and duration details will be saved as well. The active animation can then be correctly played back by the *SLIDENEO* slide show program that accompanies *NEOchrome*. It's very important not to stop cycling when you're making the save. Although this contradicts the documentation that accompanies the program, it's the only method that works.

Ramp line animation is excellent for fancy displays and for certain special effects, but color cycling also enables the *NEOchrome* user to mimic classic, Disney-type animation. This type of animation can be difficult to comprehend, but *NEOchrome* makes it so easy that it's worth the attempt. By



the way, some writers label this technique *simulated animation*, but that terminology is misleading. In this context, animation itself is simulated motion. If a drawing looks like it's really moving, then it's animated.

### Meet Ernesto

In Figure 5, you can see the creation of a cartoon character, Ernesto, the top-hatted platypus. From his basic standing pose at the top left side of the figure, three sequential imaginary running poses are developed. The copy-box tool makes the process relatively simple. The drawings are assembled into the sequence shown in the bottom half of the figure. The intention is to have Ernesto run from the right side of the picture to somewhere off the screen on the left.

**Figure 5. Drawings for a Palette Animation**

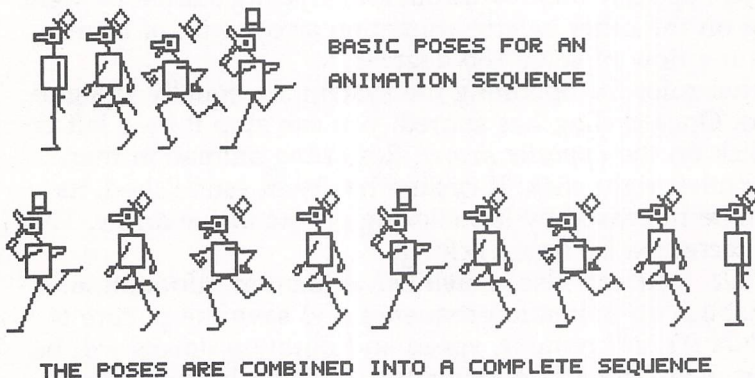
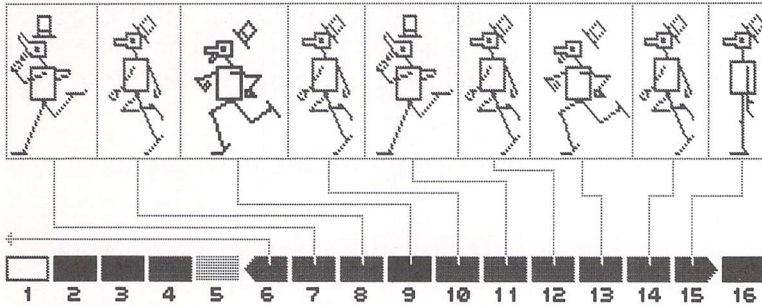


Figure 6 shows how Ernesto and the running sequence are set up to match palette positions. Ten positions, 6 through 15, are assigned to the nine drawings in the sequence. However, only one of the positions, number 9, produces a dark image of the character. The rest are drawn in what will be the eventual background color. As a result, eight of the drawings will be invisible at all times since they will simply blend into the background color. Consequently, the one dark drawing of Ernesto will appear to run across the display.

**Figure 6. Palette and Drawings Adjusted for Animation**



Position 6, as shown by the arrow, is dedicated to a non-existent part of the sequence to create a brief delay onscreen. That is, when the palette is cycled to the left—the planned direction for Ernesto's run—this provides a timing delay before the darkly colored image reappears on the right side of the screen. Tricks like this are essential to realistic animation and are not easily learned. However, working with *NEOchrome* removes a lot of the drudgery. If an animation is close to what you want, but still not quite right, save it to disk. You can then afford to experiment to your heart's content; your previous efforts are safely stored away.

### Background Details

Palette positions 1–5 and 16 can be used to build up background details of the picture, but one important factor must be kept in mind. One of these blocks must be reserved for the same color as the eight invisible Ernestos. The colors in the palette blocks between the ramp pointers are constantly changing and therefore are unsuitable for this purpose. Balancing color demands for this type of animation is a critical area of planning.

In Figure 7, a city skyline is drawn around Ernesto and filled with the background blending color—causing eight of the Ernesto figures to disappear. This is actually the tone from palette position 5 in Figure 6. You'll notice that the light



drawings of the platypus have been adjusted to match. In the bottom half of the figure, window details have been painted onto the buildings around the still visible heads and torsos of the eight pale Ernestos. As indicated by the fill icon, these will then be filled in with the background blending color from position 5.

**Figure 7. Background and Details Added to Sequence**

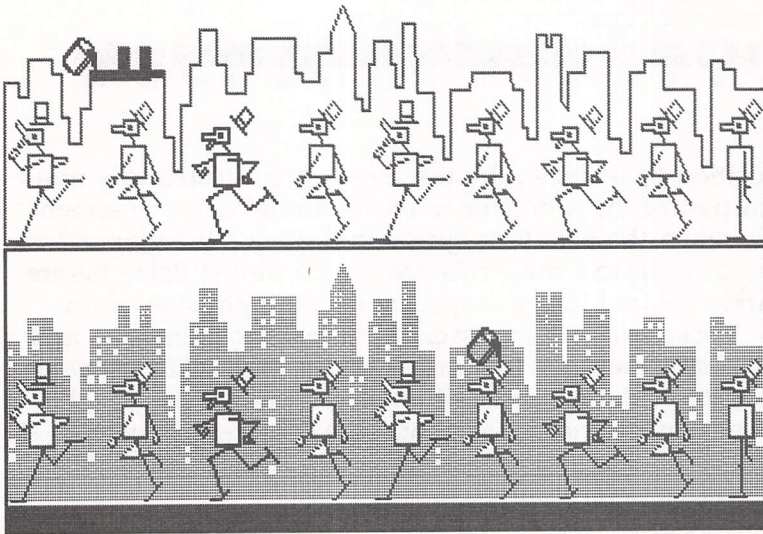
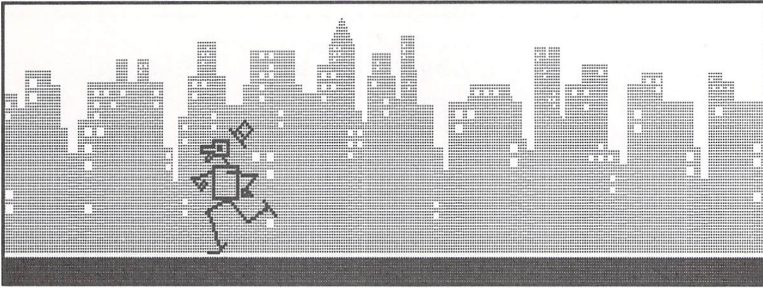


Figure 8 shows a single Ernesto, frozen in one frame of his endless run. The finished animation works successfully: As the colors cycle, eight background-color Ernestos stay invisible. The dark color, however, is cycled through all nine positions (and the tenth delay block), thus creating the impression that Ernesto is racing across the screen on a typically urgent platypus errand.

There is virtually no limit to the number of animated objects that can be simultaneously displayed by this method. On the other hand, since only one range of colors can be cycled at any point in time, there are severe limits on the colors available for animation. For example, a bird could fly parallel to Ernesto, and a streetcar could travel in the opposite direction.

**Figure 8. The Finished Animation**



However, they would have to move against the same colored background and would need to appear in Ernesto's dark color. Clever scene design can do much to overcome these limits, but animation always requires considerable work and forethought. Happily, *NEOchrome* goes a long way toward reducing such effort to enjoyably challenging levels.

When you combine palette animation with ramp lines and the excellent color chart, it becomes apparent that *NEOchrome* is a powerhouse graphics program. Contrary to first impressions, you'll find that the program has plenty of muscle under its mild-mannered exterior. And best of all, using this program's exceptional power is downright easy.



# Doodler

D. W. Neuendorf

---

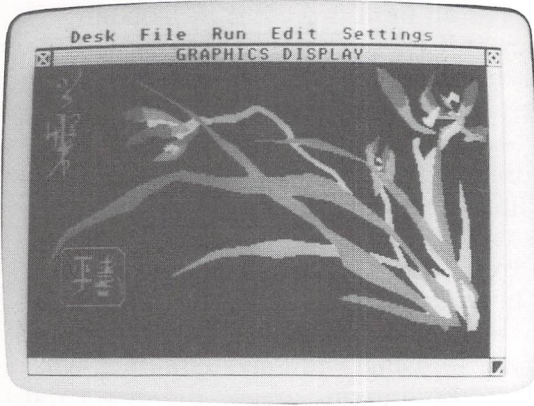
*Learn how to write a Logo program that takes advantage of GEM's built-in features and user interface. "Doodler" is for any Atari ST with Logo.*

When Atari first started shipping the 520ST in the summer of 1985, ST BASIC wasn't quite ready. The only programming language supplied was Digital Research's Logo. When I borrowed a friend's 520ST, I decided to put Logo to work in a drawing program. This version of Logo, however, has been translated from Digital's Logo for the IBM PC, and it doesn't run particularly fast on the ST. After some experimenting, I realized that it's too slow to support a full-fledged drawing utility.

Not yet ready to write off the ST/Logo combination, I considered alternatives. Calls to the operating system were out, since there was only a limited memory map and no CALL statement. Furthermore, Logo restricts the areas of memory accessible to the EXAMINE and DEPOSIT commands (similar to BASIC's PEEK and POKE). Even hand-assembled machine language routines are useless without a way to call them.

Then I remembered GEM—Digital Research's Graphics Environment Manager, which sits as a shell on TOS, the ST's operating system. In Logo, GEM has a Settings menu that lets you change line-drawing colors, line-drawing widths, fill colors and patterns, and other parameters. There must have been some reason that Atari included all these settings in a drop-down menu, duplicating many of the Logo commands. One very good reason, I concluded, was to avoid forcing someone like me to code a complex user interface in Logo, which would bog down the program. After all, the Settings menu is quite similar to the menu I planned to include in my drawing program.

Then, I took a hard look at what's *really* needed to draw pictures on the ST. If we let GEM handle the fancy features via its Settings menu, all we need is an easy way to fill areas and draw lines, circles, and boxes. Providing these functions is well within Logo's capabilities. "Doodler" is the result.



*The author created this picture with "Doodler."*

### Drawing with Doodler

To use Doodler, run Logo and type in the listing below. Be sure to save at least one copy before you try to use Doodler for the first time. Next, you should decide which screen resolution mode you wish to use. The monochrome screen gives you the highest resolution—and thus the capability of drawing finely detailed pictures—but it allows only black and white. The low-resolution mode allows 16 different colors, but there's the loss of some detail. The medium-resolution mode offers more detail than low resolution, with up to 4 colors. If you wish to draw in a mode other than the one currently selected, you must quit Logo and return to the GEM desktop, then select Set Preferences from the Options menu. After making your selection, run Logo again.

Before you begin drawing with Doodler, clear the Graphics window by typing CS and pressing Return at a Logo top-level question mark (?) prompt. Then type SKETCH. You'll probably want to expand the Logo Graphics window to full-screen size to give yourself more room to work. You can alter Doodler's pen color at any time, along with the background color, line width, fill color, and fill pattern settings. Just drop down the Settings menu and select the Graphics option. Click the pointer on the setting you want to change; then type in the appropriate number and click on the OK box.

To draw with Doodler, you connect a series of lines, end to end. Choose the beginning of a series of line segments by moving the mouse pointer to the first desired endpoint and



pressing the left button. For all button presses in Doodler, hold down the button for at least a second or two; Logo cannot detect a faster press. If a Doodler command doesn't get a response, you're probably not holding down the button long enough.

You can specify subsequent endpoints the same way. You can draw a continuous line by holding down the left button while moving the mouse. However, because Logo isn't very adept at reading the buttons, you must move the mouse very slowly to draw smooth lines. To end the series of connected line segments, move the pointer outside the drawing area and press the left button. If you're using a full-size Graphics window, the best place outside the drawing area is the upper right corner of the screen beyond the menu bar.

To fill an area with color, place the mouse pointer inside the area and press the right button. Be sure the area you're trying to fill is completely enclosed by lines. If there are any gaps, the fill "spills out" and colors the entire background.

To draw circles or boxes, press the right button while the pointer is outside the drawing area. A prompt asks you to press the C or B key for a circle or box, respectively. Pressing any other key exits the circle/box mode. After you press C to choose a circle, point to the desired center and press the left button. Then move the pointer to the desired radius and press the button again. If you press B to choose a box, point first to its lower left corner and press the left button. Then point to its upper right corner and press the button again.

You can erase portions of your drawing by dropping down the Settings menu, selecting the Graphics option, changing the line color to match the background color, then drawing over the parts you want to erase. You may also want to widen the line setting for this purpose.

### How It Works

The top-level procedure, SKETCH, does a little initializing before it invokes the main procedure, PT, which executes repeatedly. PT stores the current mouse status in the variable T, and then analyzes it for the state of the left and right mouse buttons. Each mouse button has two functions, depending on whether the pointer is inside or outside the drawing area when the button is pressed.

Pressing the left button (indicated in ITEM 3 of MOUSE) specifies the endpoints in a series of connected line segments. DRAW? sets a flag, depending on whether ITEM 5 of MOUSE is TRUE (pointer inside the drawing area) or FALSE (pointer outside the drawing area). This flag, in turn, controls whether DR draws another line segment or sets a new starting point.

Pressing the right button (indicated in ITEM 4 of MOUSE) fills an area with the current fill pattern if the pointer is within the Graphics window boundaries (ITEM 5 of MOUSE is TRUE), or, it initiates circle or box drawing if the pointer is outside the Graphics window (ITEM 5 of MOUSE is FALSE). The circle and box prompts are drawn in the pen-reversed mode; then they're selectively erased by being redrawn in the same place. Although it can be hard to read these prompts over existing screen graphics, the alternative—printing to the Dialog window—stops the program.

You can save your artwork by using the Save Pic option in the File menu. Reload previous drawings with the Load Pic option in that same menu. When you reload pictures, you must set the screen for the same resolution that was in effect when the picture was saved. For example, you cannot load a picture drawn on the low-resolution screen into a medium-resolution Graphics window.

The lesson for programmers here is that programming on the ST is very different from programming on earlier computers with traditional operating systems. Whether you're using Logo or a very fast compiled language, it would be a mistake to ignore the high-level tools available in GEM and TOS. Not only is it a waste of effort to write everything from scratch, but it's also wise to stick to the user interface that is already thoroughly familiar to every ST owner.

### Doodler in Logo

```
TO SKETCH
  HIDE TURTLE
  PENUP
  MAKE "GFILL "TRUE
  MAKE "TF Ø
  PT
END

TO PT
  MAKE "T MOUSE
  IF ITEM 3 :T [DRAW?]
```



## CHAPTER FIVE

---

```
IF ITEM 4 :T [BCORF]
PT
END

TO DRAW?
IF ITEM 5 :T [DR] [MAKE "TF 0]
END

TO DR
IF (:TF = 0)
[PENUP SETPOS :T MAKE "TF 1]
[PENDOWN SETPOS :T PENUP]
END

TO BCORF
IF ITEM 5 :T
[SETPOS PIECE 1 2 :T FILL]
[BORC]
END

TO BORC
MAKE "PCOL ITEM 5 TURTLEFACTS
BCMSG
MAKE "CH READCHAR
BCMSG
SETPC :PCOL
IF (:CH = "B) [BX]
IF (:CH = "C) [CIRC]
MAKE "TF 0
END

TO BCMSG
SETPOS [-70 80]
SETH 0
TMSG [Circle: Press C]
TMSG [Box: Press B]
TMSG [Abort: Press any]
TMSG [# # # # other key]
END

TO TMSG :MESSAGE
PENREVERSE
TURTLETEXT :MESSAGE
PENUP
BACK 18
END

TO BX
MAKE "GFILL "FALSE
BOX MBP
```

```
MAKE "GFILL "TRUE
END
```

```
TO CIRC
  MAKE "GFILL "FALSE
  CIRCLE MCP
  MAKE "GFILL "TRUE
END
```

```
TO MBP
  GETPOINTS
  MAKE "PAR3 ABS ((FIRST :PAR2) - (FIRST :PAR1))
  MAKE "PAR4 ABS ((LAST :PAR2) - (LAST :PAR1))
  OUTPUT (SENTENCE :PAR1 :PAR3 :PAR4)
END
```

```
TO MCP
  GETPOINTS
  MAKE "PAR3 (ABS ((FIRST :PAR2) - (FIRST :PAR1)))
    ^ 2
  MAKE "PAR4 (ABS ((LAST :PAR2) - (LAST :PAR1))) ^
    2
  OUTPUT SENTENCE :PAR1 SQRT (:PAR3 + :PAR4)
END
```

```
TO GETPOINTS
  MAKE "PAR1 GETPOS
  DELAY
  MAKE "PAR2 GETPOS
END
```

```
TO GETPOS
  MAKE "T MOUSE
  IF (ITEM 3 :T) [OUTPUT PIECE 1 2 :T]
  GETPOS
END
```

```
TO DELAY
  REPEAT 10 [MAKE "JUNK SIN 5]
END
```



# Making Music on the ST

David Florance

---

*“MELODYST” is an easy-to-use music-generating program. Even if you know nothing about music, you’ll quickly learn how to incorporate music into your own programs.*

The Atari ST provides users with some exciting prospects in the area of music and sound. In this article we’ll deal specifically with making music on the ST through internally processed software. The programs here are written entirely in ST BASIC. If you want to explore ST music in more depth, you might want to explore C and Pascal, which afford greater access to the ST’s capabilities.

## SOUND

First, let’s see how ST BASIC handles music. We have two useful statements at our disposal, **SOUND** and **WAVE**. The **SOUND** statement gives access to the tone generator for making songs and melodies. Its syntax is

**SOUND** *voice, volume, pitch, range, duration*

The *voice* parameter chooses one of the ST’s channels through which the note can be heard. The ST has three channels, usually referred to as A, B, and C. In programming, the numbers 1, 2, and 3 designate channels A, B, and C, respectively. The second parameter, *volume*, controls the level of the note’s loudness. The choices here are 0–15; for soft notes, use a lower number. The third parameter, *pitch*, tells the ST what note to play. As you can see from Table 1, each of the notes in the chromatic tonal scale is represented by a number, 1–12.

The ST takes the number specified for pitch and can tell which frequency to generate. However, it can play far more than just these 12 notes. With the fourth parameter, *range*, you can specify one of eight octaves. (An octave is one set of 12 pitch values. It’s called an octave because there are 7 pitches that are natural, not sharpened or flatted, before the notes start

**Table 1. Chromatic Scale**

Note Name	Number
C	1
C# or D $\flat$	2
D	3
D# or E $\flat$	4
E	5
F	6
F# or G $\flat$	7
G	8
G# or A $\flat$	9
A	10
A# or B $\flat$	11
B	12

to repeat on the eighth pitch.) Thus, the ST can play 96 ( $8 \times 12$ ) different notes. Octaves 2–6 are the most frequently used octaves for creating songs. For range, you can use a number from 1 through 8.

The final parameter, *duration*, tells the ST how long the note will be sounded. Values permitted are 0–65536. Each unit of duration is equal to approximately 1/60 second, so a duration of 60 causes the note to be sounded for about one second. Type the following statement in direct mode (without a line number at the OK prompt) and hear what happens:

**SOUND 1,15,10,4,60:SOUND 1,0,0,0,0**

The ST will play in voice 1 (channel A) at maximum volume 15, pitch 10 (note A) in a range of 4 (fourth octave) for a duration of 60 (about one second). Then it will turn off everything it has turned on by placing zeros in all the parameters for voice 1.

### WAVE

The second BASIC statement useful for music applications is WAVE. It allows you to alter the shape of the note to be sounded. The syntax is

**WAVE** *oscillator, envelope, form, period, length*

The first parameter, *oscillator*, tells the ST what kind of SOUND is wanted—either periodic (tonal), aperiodic (noise), or both. This number is put into the sound register's first six bits. Bits 0–2 are for tone, and 3–5 are for noise. Refer to Table 2.

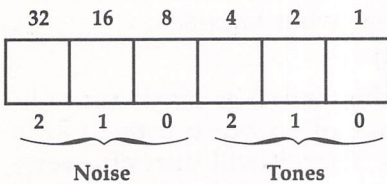


**Table 2. WAVES**

WAVE	Result
0	All voices quiet
1	Voice 1 SOUNDS tones
2	Voice 2 SOUNDS tones
3	Voices 1 and 2 SOUND tones
4	Voice 3 SOUNDS tones
5	Voices 1 and 3 SOUND tones
6	Voices 2 and 3 SOUND tones
7	All three voices SOUND tones
8	Voice 1 SOUNDS noise
16	Voice 2 SOUNDS noise
24	Voices 1 and 2 SOUND noise
32	Voice 3 SOUNDS noise
40	Voices 1 and 3 SOUND noise
48	Voices 2 and 3 SOUND noise
56	All three voices SOUND noise
63	All three voices SOUND tones and noise

Note that any combination is allowed and gives the obvious result. For example, WAVE 17 turns voice 1 to tones and voice 2 to noise. Figure 1 shows how the bits are configured.

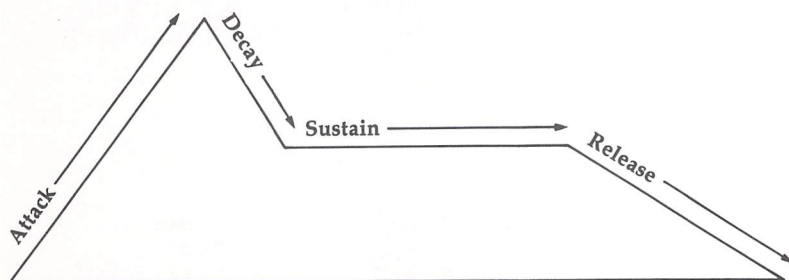
**Figure 1. Noise and Tones**



The next parameter, *envelope*, uses bits to designate which voices will be controlled by the envelope generator. Using a 1 means let voice 1 through; 3 means let voices 1 and 2 through; and 7 means let all voices through.

The parameter *form* lets you choose a waveform. All sounds take the general shape of the attack-decay-sustain-release envelope that you see in Figure 2. *Attack* determines the amount of time it takes for a note to reach its maximum, or peak, volume level. *Decay* controls how quickly a sound falls from its peak level to its sustained volume. *Sustain* is the volume at which a sound is held during the middle portion of its cycle, and *Release* is the time it takes for the sustained volume to reach silence.

**Figure 2. ADSR Envelope**



When numbers are put into the *form* parameter, the shape changes. Figure 3 shows the built-in shapes and their numbers.

**Figure 3. Waveforms**

Binary	Dec	Waveform Shape
00XX	0-3	
01XX	4-7	
1000	8	
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	



*Period* designates the time between the cycles of the form, and *length* tells the ST how long to wait before executing the next BASIC command. Type in this short program, "WAVESAMP," if you'd like to experiment with the WAVE statement.

### Program 1. WAVESAMP

```
10  ' wave sampler
15  fullw 2:clearw 2
20  input "Oscillator (1,3, or 7)      ";osc
30  input "Envelope (1,3, or 7)       ";env
40  input "Form (0-15)                 ";form
50  input "Period (0-65535)           ";per
60  input "Delay (0-65535)            ";del
80  wave osc,env,form,per,del
90  rem for x=1 to 8:for y=1 to 12:sound 1,15,y,
    x,1:next y,x
100 goto 20
```

Notice line 90 in WAVESAMP. If you take the REM out of the line, you'll hear a chromatic scale along with the wave-form you designated.

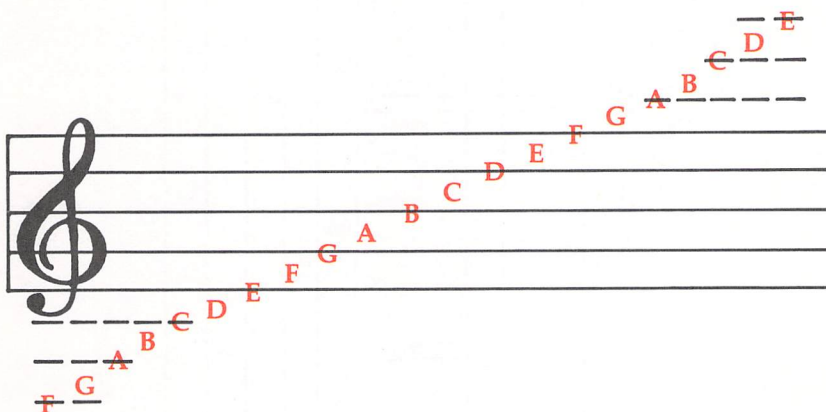
### Making Music with MELODYST

Type in the three programs at the end of this article and save them to disk. The main program, "MELODYST," is a melody editor. It uses the ST's sound generator chip to allow you to create, listen to, edit, store, and recall melodies, which can then be added to your own programs. It runs only in low-resolution mode. The other two programs are actually subroutines designed to enable you to use files created with MELODYST in your own programs. These programs are simple in construction so that they'll be easy to use.

When you run MELODYST, a command window will appear and show you six options: NAME NOTES, PLAY NOTES, LOAD NOTES, SAVE NOTES, EDIT NOTES, and GOTO BASIC. The first option, NAME NOTES, allows you to create melodies. If you're familiar with the traditional tonal scales, you already know how to use this option. Simply type in the

note value (pitch) you desire along with its octave and duration. If you're not familiar with musical staves and clefs, you can use Figure 4 to transcribe sheet music to your Atari ST.

**Figure 4. Musical Notes**



Look at each note on your sheet music, find its corresponding value in Figure 4, and type in the name of the note when you're prompted for PITCH. If there's a sharp ( # ) or flat ( b ) beside the note or anywhere in the music, you must add that to the note value. Here's a point to remember: If a line goes through the middle of a note head (the circular part of the note), the note is said to be on that line. If not, it is said to be on a space. The program MELODYST and the computer handle sharps only. You can use Table 3 to convert all flats to sharps.

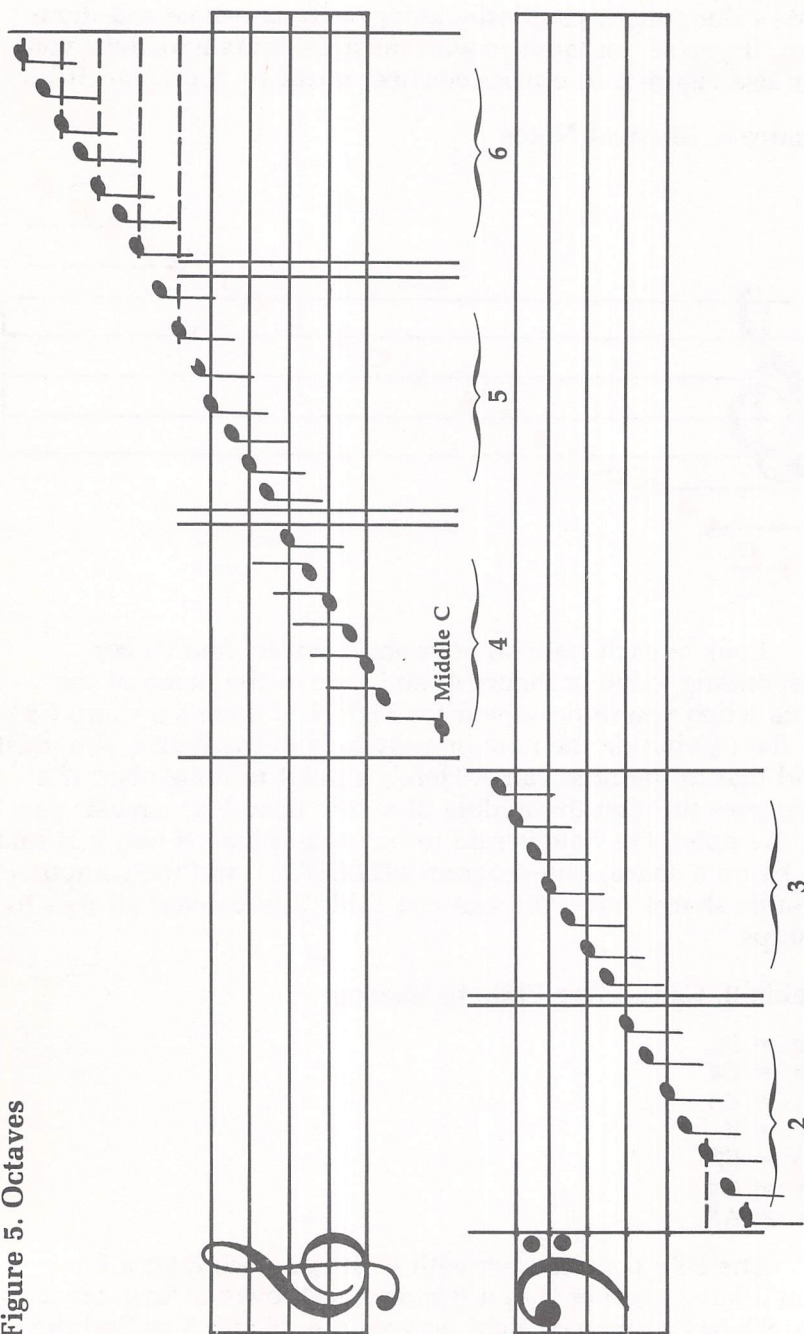
**Table 3. Converting Flats to Sharps**

C $\flat$	= B
D $\flat$	= C $\sharp$
E $\flat$	= D $\sharp$
F $\flat$	= E
G $\flat$	= F $\sharp$
A $\flat$	= G $\sharp$
B $\flat$	= A $\sharp$

The ST's octaves start with C. If you encounter a C $\flat$ , you'll have to enter it as a B in the next lower octave. Since the ST has a range of eight octaves, use Figure 5 to find the



Figure 5. Octaves












## CHAPTER FIVE

---

octave of the note you want. Notice that not all octaves are given. The octaves that are not shown are used only infrequently for melodies. Find the octave in which your note falls, and enter that value for OCTAVE. For example, if you want to enter a flatted middle C, enter B in octave 3. Then refer to Table 4 to enter the time allowed for your note to sound.

**Table 4. Note Duration**

Note	Note Name	Duration
	Thirty-second	2
	Sixteenth	5
	Eighth	10
	Quarter	20
	Dotted quarter	30
	Half	40
	Dotted half	60
	Whole	80
	Dotted whole	120

When you've determined the time value, enter that number for DURATION. Don't worry if you're not sure how fast or slow your piece should go. MELODYST lets you adjust that later. Follow the three steps above until you have entered your melody in full. Remember that on sheet music the top note is almost always the melody.



### Listening to Your Creation

When you choose the next option, **PLAY NOTES**, you can hear the notes you've entered. **MELODYST** allows variance in the general speed of the piece you've entered. In musical terms this is known as the *tempo*. You can choose a tempo from 1 to 255, with 1 being the least amount of delay between notes, and 255, the most. Specific control is combined with the note values you have designated. In other words, the larger the note values that you have entered are, the more the tempo will alter the speed of the piece. Table 5 gives suggestions on tempo. The range is from *presto* (quick) to *lento* (slow).

**Table 5. Tempo**

<b>Designate</b>	<b>Tempo</b>
Presto	1-5
Vivace	6-15
Allegro	16-30
Moderato	31-50
Andante	51-80
Adagio	81-120
Largo	121-170
Lento	171-255

When you've determined the tempo of your piece, enter a number between 1 and 255, inclusive, to respond to the **TEMPO** prompt. Next, **MELODYST** will ask you for a **WAVEFORM**. Choose a number between 0 and 14, inclusive, for one of the **ST**'s built-in shape generators. Experiment with different values for different sounds. There is one more listening parameter, called **PERIOD**, which controls how fast each note cycles (it may not be audible). Experiment until you find one you like.

When you're through listening, a dialog window will appear asking whether you want to hear the piece again. The third and fourth options, **LOAD NOTES** and **SAVE NOTES**, allow you to load and save the melody to disk. The option called **EDIT NOTES** allows just that—you can change the parameters for any note you have created. The last option, **GOTO BASIC**, returns you to **ST BASIC** when you've finished the session.

As you continue using **MELODYST**, you'll find that making music becomes less and less laborious. If you know little about music, **MELODYST** can double as a tutorial. Don't be

surprised if you become more adept at reading music by using the program.

### Music in Your Own Programs

It's easy to use songs created with MELODYST in your own programs. Program 3, "MELOADER," is a subroutine which can be inserted into any ST BASIC program. Call the subroutine, replacing the f\$ in line 5000 with the name of your MELODYST file. Make sure your song file is on the same disk as your program, and this routine will load the notes you have saved into the variables *p%(nn)*, *r%(nn)*, *d%(nn)*, and *e\$(nn)*.

Program 4, "MELPLAY," also is a subroutine that you can insert into any ST BASIC program. It takes the variables loaded with MELOADER and uses the SOUND statement to play through them. Be sure to specify a value for the variable *de*, which is inside a delay loop and controls the tempo. If you don't do this, your song may play faster than you would like. Call the subroutine, and the music will be played. You'll need to use both MELPLAY and MELOADER (or similar routines that you may write) for this process to achieve the desired results. Note as well that in line 10 MELOADER DIMs the variables needed. If your program contains a line 10, to avoid conflict, RENUMber your program before MERGEing the subroutine into your program, or RENUMber MELOADER's line 10.

### The Sound Generator

The Atari ST is equipped with the Yamaha YM-2149, which is a member of the same family of sound chips as the GI AY-3-8190. These programmable sound generators (PSGs), some of which were used in earlier micros and were developed for arcade games, are 16-register synthesis sound chips. When accessed from GEM, the YM-2149 displays impressive power for generating and altering sounds. It has three independently programmable voices, a programmable tonal pitch and white noise mixer, two 8-bit data ports, analog output, a D/A digital converter, software controllable envelopes for attack, decay, sustain, and release, and an amplitude control block for volume output. All these are top-quality attributes that demonstrate just how well the Atari ST is put together. While it is impracticable to seek full use of this dynamic chip through



## CHAPTER FIVE

BASIC, it is well-suited for use with other languages. If you want to explore the ST's sound capabilities in greater depth, you might consider C or Pascal. You can see, even through simply conceived BASIC programs like MELODYST, that the YM-2149 is quite versatile.

### Program 2. MELODYST

```
10      rem MELODYST
20      openw2:fullw 2:clearw 2:gosub 9600
22      if peek(systab)<>4 then print "This program
      requires LOW RESOLUTION.":end
24      print "This program politely ignores"
25      print "you if you don't enable the"
26      print "CAPS LOCK. Please make sure"
27      print "it is on."
28      for f=1 to 5000:next:clearw 2
30      dim n(255),p%(255),r%(255),d%(255),e$(255)
35      nn=1:er$=""
40      for x=1 to 6:read a$(x):next
200     ' main loop
210     gosub 3000
220     if en=1 then closew 2:closew 1:end
240     goto 200
500     ' name opt
505     nn=1:gosub 9500
510     clearw 2:color 1,4,1,6,2:fill 2,2
520     gotoxy 6,4:print er$:gotoxy 6,4:print "Note
      number ";nn
530     gotoxy 6,5:print er$:gotoxy 6,5:input "Pitch
      ";p$
540     n$(1)=left$(p$,1):n$(2)=right$(p$,1):p%(nn)=
      asc(n$(1))-64
550     if p%(nn)<1 or p%(nn)>7 then gosub 520
560     e$(nn)=p$:gosub 7000
565     if len(p$)>2 then 520
567     if n$(2)="#" then p%(nn)=p%(nn)+1:if p%(nn)=
      13 then 520
569     gotoxy 6,6:print er$
570     gotoxy 6,6:input "Octave ";r%(nn):if r%(nn)<
      1 or r%(nn)>8 then 570
580     gotoxy 6,7:print er$
590     gotoxy 6,7:input "Duration ";d%(nn)
600     gotoxy 6,9:print er$
602     gotoxy 6,9:input "More <Y/N>";r$
610     if r$="N" then 640
630     nn=nn+1:goto 510
640     return
700     en=1:return
```

```
1000 ' save opt
1005 fullw 2:clearw 2:color 1,2,1,1,1:fill 10,10
1010 gotoxy 15,1:print "SAVE"
1015 gotoxy 10,3:input "Filename";f$
1016 f$=mid$(f$,1,10):gotoxy 10,5
1020 print "Save ";f$
1030 gosub 9200:gosub 8100
1035 if peek(gt)=2 then return
1040 if peek(gt)=1 then 1045
1045 open "0",#1,f$
1050 for x=1 to nn:write #1,p%(x),r%(x),d%(x),e$(
x)
1060 next
1070 close #1
1999 return
2000 ' load opt
2001 gosub 9500:on error goto 9400
2005 fullw 2:clearw 2:color 1,2,1,1,1:fill 10,10
2010 gotoxy 15,1:print "LOAD"
2015 gotoxy 10,3:input "Filename";f$
2016 f$=mid$(f$,1,10):gotoxy 10,5
2020 print "Load ";f$
2030 gosub 9200:gosub 8100
2035 if peek(gt)=1 then 2045
2040 if peek(gt)=2 then return
2045 x=0:open "I",#1,f$
2046 on error goto 0
2047 while not eof(1)
2049 x=x+1
2050 input #1,p%(x),r%(x),d%(x),e$(x)
2070 wend
2080 close #1
2999 nn=x:return
3000 color 1,3,1,1,1:clearw 2:fill 10,10:rem init
3005 for x=1 to 6:gotoxy 7,x+5:print x;">>>>>>>>"
"a$(x):next
3010 gotoxy 12,3:print "COMMAND MENU"
3015 gotoxy 11,15:print "CLICK TO CHOOSE"
3020 gosub 8000:if button=0 then 3020
3030 if x<64 or x>240 then 3020
3040 if y<78 or y>130 then 3020
3050 if y<86 then ch=1:goto 3060
3052 if y<94 then ch=2:goto 3060
3054 if y<103 then ch=3:goto 3060
3056 if y<112 then ch=4:goto 3060
3058 if y<121 then ch=5:goto 3060
3059 if y<130 then ch=6
3060 on ch gosub 500,6000,2000,1000,5000,700
3099 return
```



## CHAPTER FIVE

```
4000 rem print out incorrect value message
4499 return
4500 gotoxy 15,7:print "
4999 return
5000 ' edit opt
5005 fullw 2:clearw 2
5010 gotoxy 6,5:print "EDIT"
5020 for es=1 to nn
5046 print "Note #"es" "e$(es):next
5050 gotoxy 0,16:print "Which note to change";:in
put r$:r=val(r$)
5052 if r<1 or r>nn then gotoxy 0,16:print er$:go
to 5050
5054 clearw 2:gotoxy 6,6:print "Note # "r
5056 gotoxy 6,7:print "Pitch "e$(r):gotoxy 6,8:pr
int "Octave "r%(r)
5058 gotoxy 6,9:print "Duration "d%(r)
5060 gotoxy 6,10:input "New Pitch";np$
5062 n$(1)=left$(np$,1):n$(2)=right$(np$,1):p%(r)
=asc(n$(1))-64
5064 if p%(r)<1 or p%(r)>7 then 5005
5066 e$(r)=np$:f1=1:gosub 7000
5068 if n$(2)="#" then p%(r)=p%(r)+1:if p%(nn)=13
then 5005
5089 gotoxy 6,11:input "New Octave";nr$
5090 gotoxy 6,12:input "New Duration";nd$:clearw
2:gotoxy 6,7
5092 print "Note # "r:gotoxy 6,8:print "Pitch "np
$:gotoxy 6,9
5094 print "Octave "nr$:gotoxy 6,10:print "Durati
on ";nd$
5095 gotoxy 6,12:input "OK";ok$:if ok$<>"Y" then
5005
5097 nr=val(nr$):r%(r)=nr:nd=val(nd$):d%(r)=nd
5098 gosub 9300:gosub 8100:if peek(gt)<>2 then 50
05
5099 return
6000 ' play opt
6005 clearw 2:color 1,6,1,1,1:fill 10,10
6006 gotoxy 6,1:input "Tempo (1-255)";de:if de<1
or de>255 then 6006
6007 gotoxy 6,2:input "Waveform (0-14)";wv:if wv<
0 or wv>14 then 6007
6008 gotoxy 6,3:input "Period (0-4096)";pe:if pe<
0 or pe>4096 then 6008
6009 gotoxy 6,5:print "Playing your creation..."
6010 wave 1,1,wv,pe:for pn=1 to nn
6020 sound 1,12,p%(pn),r%(pn),d%(pn)
6025 for j=1 to de:next j:sound 1,0,0,0,0
6030 next:sound 1,0,0,0,0:gosub 9100:gosub 8100
```

```
6040 if peek(gt)=1 then 6010
6099 return
7000 nt=nn:if f1=1 then f1=0:nt=r
7002 on p%(nt) gosub 7010,7020,7030,7040,7050,706
0,7070:return
7010 p%(nt)=10:return
7020 p%(nt)=12:return
7030 p%(nt)=1:return
7040 p%(nt)=3:return
7050 p%(nt)=5:return
7060 p%(nt)=6:return
7070 p%(nt)=8:return
8000 ' mouse check
8010 poke contrl,124
8020 poke contrl+2,0:poke contrl+6,0
8030 vdisys(0)
8040 x=peek(ptsout):y=peek(ptsout+2)
8050 button = peek(intout)
8060 return
8100 gb#=GB
8110 gn=PEEK(gb#+8)
8120 gt=PEEK(gb#+12)
8130 ad#=PEEK(gb#+16)
8140 db=1
8150 POKE gn ,db
8180 POKE ad#,VARPTR(msg$)
8190 GEMSYS(52)
8199 return
9000 data NAME NOTES,PLAY NOTES,LOAD NOTES,SAVE N
OTES,EDIT NOTES,GOTO BASIC
9100 msg$="[2][!Play it again?!]"
9110 msg$=msg$+"[Yes!No]" +CHR$(0)+CHR$(0)
9120 return
9200 msg$="[3][!Please confirm.]"
9210 msg$=msg$+"[OK!Cancel]" +CHR$(0)+CHR$(0)
9220 return
9300 msg$="[2][!More Editing?!]"
9310 msg$=msg$+"[Yes!No]" +CHR$(0)+CHR$(0)
9320 return
9400 gb#=gb
9410 gn=peek(gb#+8):gt=peek(gb#+12)
9420 er=2:poke gn,er
9430 gemsys(53)
9450 resume 200
9500 ' clear variables
9510 for c=1 to 255:p%(c)=0:r%(c)=0
9520 d%(c)=0:e$(c)="":next
9530 return
9600 gb# = gb
```



## CHAPTER FIVE

---

```
9605  gn=peek(gb#+8)
9610  poke gn+0,peek(systab+8)
9615  poke gn+2,2
9620  st# = gn+4:gosub 9700
9625  msg$= msg$+chr$(0)
9630  poke st#,varptr(msg$)
9635  gemsys(105)
9640  return
9700  msg$="MELODYST"
9710  return
```

### Program 3. MELOADER

```
10    dim p%(255),r%(255),d%(255),e$(255)
5000  nn=0:open "I",#1,f$:" where f$ is name of yo
      ur music file
5010  while not eof(1)
5020  nn=nn+1
5030  input #1,p%(nn),r%(nn),d%(nn),e$(nn)
5040  wend
5050  close #1
5060  return
```

### Program 4. MELPLAY

```
5100  for pn=1 to nn
5110  sound 1,12,p%(pn),r%(pn),d%(pn)
5120  for j=1 to de:next j:sound 1,0,0,0,0
5130  next:sound 1,0,0,0,0
5140  return
```

## CHAPTER SIX

---

# C Programming





# Introduction to C Programming

Sheldon Leemon

---

*The C language's portability, structured design, and reusable functions are making it increasingly popular as an alternative to BASIC. A comparison of a program written in BASIC and in C shows you similarities and differences.*

As recently as 1984, the C programming language was virtually unknown to microcomputer owners. The only book available on the subject was *The C Programming Language*, coauthored by Brian Kernighan and Dennis Ritchie, who invented C. And *K & R*, as programmers refer to it, made no mention of C programming on microcomputers (you can tell that serious programmers wrote it, because the book starts with Chapter 0). But though the Kernighan and Ritchie book is still considered the standard reference, today you can walk into almost any well-stocked bookstore and choose from dozens of books on every aspect of C programming. Most of these books concern C programming on microcomputers.

## C and Microcomputers

Despite its recent surge in popularity, C is not a new programming language. It was designed at Bell Laboratories in 1971 for the purpose of implementing the UNIX operating system on mini- and mainframe computers. Clearly, though, something has happened in the microcomputer industry recently to push this language into a new prominence. That "something" is the evolution of the personal computer from a low-priced novelty into a serious tool.

The first personal computers had 8-bit processors that ran at a clock speed of 1 or 2 MHz (megahertz). By current standards, they contained a very limited amount of RAM; 64K was the absolute maximum. And disk storage was neither very large nor very fast. All of this changed with the advent of a new generation of more powerful (and expensive) machines



like the IBM PC and the Apple Macintosh. Suddenly, there were personal computers that had 16- and 32-bit processors running at clock speeds of 4 and 8 MHz with a minimum of 128K or 256K of memory and enormous disk storage capacities on cheap hard disks. The question then became, Can the software keep up with hardware developments?

While some machine language programs like *Visicalc* were state-of-the art on the first personal computers, owners of the new PCs began to expect large, full-featured programs like Lotus' 1-2-3. Just as these programs were much more powerful than their predecessors, so too were they larger and more difficult to create. No longer could a single programmer sit down alone and expect to create a state-of-the art program in assembly language. The common pattern for developing this sophisticated type of software was to have a number of programmers working on different parts of a program. Because of the size and complexity of the programs, and the number of programmers involved, assembly language became less and less a viable alternative for the development of commercial programs.

As a result, more microcomputer programmers have turned to C as the development language of choice. C has a number of features that make it particularly well suited for developing the kind of large, complex commercial programs that PC owners expect. As with most high-level languages, developing and maintaining C programs is easier than writing them in assembly language. Since C is not as sophisticated or complex as many other high-level languages, however, it produces relatively compact programs that execute fairly quickly. And, because C has fewer restrictions than other high-level languages, it's possible to program very close to the hardware level where necessary. Programs written in C interact well with machine language programs, which makes it possible to write speed-critical portions of a program in assembly language.

### A Compiled Language

C is a compiled language that produces stand-alone machine language programs, rather than the semi-interpreted *pseudo-code* that some other high-level compiled languages produce. Not only do such programs tend to run more quickly, but they also may be sold more freely, since there usually is no need for the developer to license a "runtime package" that is

needed to execute the program. In fact, some C compilers produce actual assembly language source code, which is then assembled as handwritten programs would be. Programmers can fine-tune this source code to produce the best performance.

Since C was developed for writing operating system programs, it stands to reason that one of C's features is the degree to which it allows programmers to maximize the performance of a program. Because operating system programs control all of the computer's operations, the overall performance of a computer depends on the performance of its operating system. The proof of the excellent performance of C programs can be found in the ST's own GEM operating system, which was developed largely in C.

### **The Power of a Structured Language**

Besides good performance, C offers a number of other features that make it attractive for program development. It is a structured language, which encourages programmers to develop large programs based on a number of small, general subprograms. There are many advantages to this type of programming. First, programmers are able to narrow their scope, so they have to work with a relatively small amount of code at a time. Second, programmers can develop "libraries" of functions, which are usable in other programs. For example, if a programmer finds that the same type of data entry screens are often used in programs, he or she might write a general function to set up such screens and include this function in all programs, thus sparing duplication of effort. So, while C has a very small set of built-in functions, its ability to include libraries of C or assembly language functions makes it, in effect, highly extensible. It enables programmers to write any new "commands" that they may want, and they can then use these new commands as if they were part of C. In fact, it's possible to assemble a number of these functions into a new "language" that provides commands to perform highly specialized tasks. Many commercial vendors offer C extension packages that provide complete libraries of database functions, telecommunications functions, graphics functions, and the like. You can buy the skeleton of a working program and add your own user interface rather than having to develop the whole thing from scratch.



### Portability of C

Not only can the C functions that you create be used in programs other than the one for which they were originally created, but frequently they can be used on other computers as well. That is because C is a fairly portable language. Although no absolute standard has been defined for the language, in practice, most implementations are quite compatible with one another. Almost all adhere to the basic definition of the language laid down in the Kernighan and Ritchie book. And many implementations have standardized on the extensions added to the language by the various official UNIX releases. The only area where there are major hardware-related differences is in input/output operations, particularly in the graphics environment. Obviously, the portions of programs dealing with icons, drop-down menus, and the like, will not easily be transported from the ST to other computers (unless they happen to run the GEM operating system). But as long as these functions are isolated in their own subprograms, the amount of hardware-specific translation work that must be done to get the program running on another machine can be kept to a minimum.

### Writing C Programs

Some BASIC programmers are intimidated by C's image as a difficult language to master. Yet once you get used to its distinctive syntax and style, it's possible to begin writing C programs in just a short time. Perhaps the best way to demonstrate that C isn't so mysterious is to show a C program. We'll look at two program listings, one in C and one in BASIC. Each produces a list of the prime numbers from 2 to 50. If your math needs a refresher, *primes* are numbers that are not evenly divisible by any number. The list that is produced looks like this:

```
2
3
5
7
11
..
..
47
```

First, here's the C language version:

```
/* Sieve.c —Finds the prime numbers from 2 to SIZE */
#include <stdio.h>
main()
{
    int num, x, count;                /* declare & initialize variables */
    #define SIZE 50
    char flags[SIZE+1];
    num = 2;

    for (x = num; x <= SIZE; x = x+1)
        flags[x]=1;                  /* set all flags */
    while (num <SIZE/2)
    {
        for (x = 2*num; x <= SIZE; x = x+num)
            flags[x] = 0;            /* eliminate multiples */
        num = num +1;
    }
    for (x = 2; x <= SIZE; x = x+1)
        if (flags[x])
            printf("%2d \n",x); /* print the primes */
}
```

Now, here is the same program written in ST BASIC:

```
100 REM Sieve.bas—Finds the primes between 2 and size
110 '
120 DEFINT a-z          'declare and initialize variables
130 size = 50
140 DIM flags(size+1)
150 num = 2
160 '
170 FOR x = num TO size
180 flags(x) = 1        ' set all flags
190 NEXT
200 '
210 WHILE (num < size/2)
220 FOR x = 2*num TO size STEP num
230 flags(x)=0          ' eliminate all multiples
240 NEXT x
250 num = num+1
260 WEND
270 '
280 FOR x=2 TO size
280 IF flags(x) THEN PRINT USING "##";x 'print the primes
300 NEXT x
```



### BASIC and C

As you can see, the two programs are not all that different. Let's compare them statement by statement. To begin with, the C program has no line numbers. A single statement can take up one line or many lines. The compiler doesn't get confused, since each statement in C ends with a semicolon, and multiple statements that are grouped together in a single block are enclosed in braces—{ }. The arrangement of statements on the line is to a large extent left up to the individual programmer. The programmer then can do whatever is necessary to make the program neat and readable. As you can see, the various parts of the C program are grouped together in a way that makes them visually distinct.

The first statement, which starts with the characters `/*`, is a remark. It corresponds to the REM statement in line 100 of the BASIC program. In C, the remark can extend over many lines until the closing `*/` characters. It's especially important to include many remarks in a C program, since the language is compact and each statement can do a lot of work. Without comments, you may find it difficult to remember what a line of C code actually does.

Next comes the line `main()`. This marks the start of a function named *main*. All C programs are composed of functions, which are small subprograms. Every C program has at least one function called *main*, and this is where program execution begins. The parentheses after *main* show that it is a function. Some functions use values, called *parameters*, that are passed to them by other functions; these functions will contain the names of the variables listed within the parentheses. But since `main()` is the first function to execute, other functions can't pass it any values. Its parentheses are empty.

After the name of the function is a brace—{. Braces are plentiful in C programs. They mark the beginning and end of function definitions and the beginning and end of compound statements within a function. As you see here, most programmers use different levels of indentation to help visually match up left braces with their corresponding right braces.

Three statements come after the initial brace:

```
int num, x, count;           /* declare & initialize variables */
#define SIZE 50
char flags[SIZE+1];
```

The first is roughly equivalent to the `DEFINT` statement in line 120 of the BASIC program. It declares that variables named *num*, *x*, and *count* are integers. To tell the truth, though, the BASIC line is more for instruction than for function. BASIC is not a strongly typed language: Most of the time, you don't have to worry whether a simple numeric variable should be stored internally as an integer or as a floating-point value (although most BASICs give you the option to specify which will be used when necessary). Unless you specifically declare the storage type you want used with one of the `DEF` statements, BASIC assumes a default type and goes its merry way. With C, however, declaration statements are *not* optional. You must take responsibility for deciding how much storage space is allotted for each variable. You can even specify where exactly in memory information is stored. So, whenever you want to use a variable in C, you must declare ahead of time whether it will be stored as a long or short integer, single- or double-precision floating point, or as text characters. These declarations are usually made in a block at the top of the function definition.

Of course, BASIC does require that you declare the size of a subscripted variable array. In this respect, the `DIM` statement in line 140 of the BASIC program is very similar to the C declaration of the *flags* array.

The middle statement in this trio of C lines is somewhat more complicated to explain. Where the BASIC program assigns the value 50 to a variable called *size*, the C program uses the `#define` statement to define a macro called *SIZE* as the number 50. The C language possesses a facility known as the *preprocessor*. This allows programmers to define terms which will be replaced by the specified terms when they are found in the program. In these two programs, we use the terms *size* and *SIZE* to refer to the size of the group of numbers in which we are looking for primes. This makes it easy to change the size of the group; we need only to change the value of the *size* term. In BASIC, the *size* value must be assigned to a variable, even though its value stays constant; in BASIC, that's the only symbolic way to represent a number. But in C, we can use the `#define` operator to define a symbolic value. All this means is from that point on in the program, every time the compiler sees the word *SIZE*, it will substitute the number 50. We can assign a symbolic meaning to *SIZE*, which makes the program



easier to read, without having to waste storage space in our program by creating a variable for this constant value.

### Differences in BASIC and C

When we compare the bodies of the two programs, we find that there are only small differences. The first is that the form of the *for* loop used by each language is somewhat different. The BASIC format contains the starting condition of the loop, the terminating condition, and the increment condition separated by the words *TO* and *STEP*. The increment condition may be left out, in which case it is assumed that the loop variable will be increased by one each time through the loop. In C, the three conditions are enclosed in parentheses and are separated by semicolons. Although, in this particular program, each condition is related to the variable *x*, it is interesting to note that in C, unlike BASIC, the three conditions do not all have to relate to the same variable. We could declare a loop that begins with setting the value of *y* to 0, and ends when *z* is equal to 50.

The second difference is that BASIC uses the *NEXT* statement to mark the end of a *FOR* loop, while C expects the loop to consist of either a single statement or of a compound statement enclosed in braces. This compound statement may be composed of any number of lines. The same is true of the *if* conditional statement. The compound *if* statement may stretch over several lines, unlike BASIC, where *IF* must take up only one line. Likewise, where BASIC uses the *WEND* statement to define the end of the *WHILE* statement, C accomplishes the same thing by enclosing the whole body of the *WHILE* statement within braces.

Another difference is the way in which the results are printed. The C program uses a function called *printf()*. This is not part of the language proper, but is part of the standard library of I/O routines. It's an example of a function that takes parameters. The text and variables that the function operates upon appear within the parentheses that follow the function name. The *printf* function performs roughly the same task as BASIC's *PRINT USING* statement. The *%* and *d* characters specify that a decimal number is to be formatted. The number 2 specifies that the numbers are to be printed with digits before the decimal place, but none after. BASIC's *PRINT USING* template *##* does roughly the same thing. The C

*printf()* function allows for multiple substitutions, while separate BASIC statements are required for each formatted column.

### C's Special Features

The example C program should make it pretty clear that, once you get past the formal requirements of function names, braces, and declaration of variables, C is not as strange as you might have thought. Of course, C is not just BASIC in another guise. It has a number of powerful features that distinguish it quite clearly from BASIC. But, thankfully, there are enough similarities that a beginning programmer can produce working code right away and then learn to take advantage of C's special features a few at a time.

For most BASIC programmers, C's added features will be quite welcome. For example, C has a multitude of powerful math and logic operators. The statement  $x += num;$  may be less recognizable than  $x = x + num;$  but over the course of a long program, it makes for a lot less typing. In C, you can use either form. There are a number of features in C that let you pack a lot into one line. For example, you can make multiple assignments using the  $=$  operator. The statement

```
a = b = c = d = 0;
```

is just fine in C. Assignments can be made to a value that is the result of a function as well as to a constant value, as in the statement

```
a = b = c = d = getchar();
```

where *getchar()* reads in the character from the keyboard. You can even make assignments at the same time you make comparisons. For example, the statement

```
if ( (a = b) < c ) DoThis();
```

first assigns the value of *b* to *a*; then it compares that value to *c* and calls the function *DoThis* if the new value of *a* is less than that of *c*.

### More Compact

Admittedly, the C program above was, to some extent, written to look as much as possible like the BASIC program. Here is another version that is a bit more C-like:



## CHAPTER SIX

---

```
/* Sieve1.c —Finds the prime numbers from 2 to SIZE */
#include <stdio.h>
#define SIZE 50
main()
{
    int num = 2, x, count;          /* declare & initialize variables */
    char flags[SIZE+1];
        for (x = num ;x <= SIZE; x++)
            flags[x]=1;              /* set all flags */
        while (num++ <SIZE/2)
            for (x = 2*num; x <= SIZE;x += num)
                flags[x]=0;          /* eliminate multiples */
        for (x = 2; x<= SIZE; x++)
            if (flags[x])
                printf("%2d \n",x); /* print the primes */
}
```

We've taken several C shortcuts here. First, the variable *num* is assigned a value in the line in which it is declared. As stated earlier, you can assign a value to a variable just about anywhere. Also, we use the `+=` operator. Finally, we use the `++` increment operator in three places. With this operator, we can say `x++` instead of `x=x+1`. Note also that the `++` operator can be used to increment one of the variables being compared as part of the condition of the *while* statement. The `++` after the variable *num* means that after the comparison has been made to determine whether the *while* loop should continue, the value of the variable is increased by one. If the `++` came before the variable name, its value would be increased before the comparison was made.

### The C Environment

One aspect of C programming that BASIC programmers will need to adjust to is the C program development environment. The process of writing a C program is very different from that of writing a BASIC program. C is a compiled language, which means that you must first create a text file containing the program instruction. This is called the *source code*. Since C does not come with a built-in text editor as BASIC does, you must use a separate word processor or text editor program to create the source. Next, you must run the compiler program, which takes the program instructions from the source code file and

creates a new file containing a series of machine code instructions called the *object code*. Sometimes the compiler is broken down into two or three programs, each of which must be run in order to perform some intermediate step in the compilation process. Even then, the job is not complete. After the source code is compiled into object code, you must run a *linker* program. It combines your program with the code for the library functions that the program uses and thus creates an executable file (one whose name has the extender .PRG, .TOS, or .TTP). Errors may occur at any phase in the process because of faulty coding or even simple typographical errors. When they happen, you must generally start over, load your text editor, and check the source code for errors.

### Streamlining C

Programming in C is obviously more complicated than typing in BASIC program lines and then entering RUN. There are a couple of important ways, however, in which you can automate and streamline the process. The first is by using a program that gives you batch processing capabilities. These programs are generically known as DOS shells, and a number of software manufacturers sell inexpensive versions.

DOS shells provide the command-line type interface of operating systems like MS-DOS and CP/M. Instead of clicking on the icon of the program you want to run, you type a command like "Compile myfile". The batch file processing capabilities of these DOS shell programs enable you to create a text file that contains the commands necessary to run the compiler and linker programs in the order needed to create an executable program. When you run the batch file, all the steps for turning your source code into a running program are taken care of, automatically. You're then free from having to perform the many intermediate steps each time you want to compile your program.

A more sophisticated program-creation utility called a "Make" program can also be used to accomplish the same thing. Some compilers, like the *Megamax* C compiler, come with program-generation facilities built in. These make the process of moving from the edit phase to the compile-and-link phase (and back again) almost completely automatic.

The second way to reduce the time needed to create a C program is by using a RAM disk. A RAM disk program sets



aside an area of the computer's memory for use as the electronic equivalent of a disk drive. You can use this device, just like its physical equivalent, to load and save programs and data files, but it provides nearly instantaneous transfer of data. Using a RAM disk is almost like having the editor, compiler, linker, and source code all resident in memory at once, instead of your having to load them in from disk each time you need them. This cuts down the time wasted between each stage of the process to the point where you can go from source code to a working program in just a few seconds. With half a megabyte or a megabyte of memory to work with, the ST computers give you lots of room for big RAM disks. There are plenty of RAM disk programs available, both from commercial vendors and from sources in the public domain.

The C programming language is far too large a subject to allow us to explore every detail in this introduction. But as with any big task, you can learn C one step at a time. Once you try C programming, you may discover that taking full advantage of the power of your Atari ST need not be as difficult as you once may have thought.

# Moving Objects in C

Charles Brannon

---

*Two C programs demonstrate that sprite simulation and animation on the ST is possible without using the advanced techniques of machine language.*

Animation is an essential part of a computer's capabilities. Of course, we couldn't play games on a machine without movable objects, but the very nature of the ST's visual operating system depends upon the ability to move objects (the cursor, icons, sliders, windows) around the screen.

Movable objects are often implemented with special *sprite* display circuitry. However, the ST video hardware doesn't have any provision for sprites. On machines like the Atari 400/800/XL/XE, Commodore 64, and Amiga, sprites greatly simplify game programming (or any programming that employs movable objects). Sprites exist on a separate video plane, so they don't interfere with an underlying background display. Since the video hardware merges the sprites with the video at hardware speed, sprites can be moved quickly without tying up the microprocessor. On the other hand, the 68000 has power to spare; it can easily simulate sprites by virtue of its high-speed memory-moving capabilities.

## ST Sprites?

The best way to simulate sprites on the ST is to write your own routines in machine language. Yet our two example programs are written entirely in C, using only documented operating system routines. The core of the animation is based on a function called *vro\_cpyfm()* (for VDI Raster Opaque Copy Form), which can be found in the Virtual Device Interface (VDI) library. It's used to copy a rectangular block from one area of memory to another; it can be used to copy one part of the screen to another or to copy a shape from a memory buffer to any part of the screen. GEM uses this function to display icons and other screen objects.

These memory buffers are supported through a C language structure called a *memory form definition block*, or MFDB. The contents of an MFDB include a pointer to the memory



containing the shape data; variables specifying the width, height, and number of bit planes (range of allowable colors) in the shape; as well as a flag specifying whether the format of the shape data conforms to the GEM standard or is machine-specific, using the same memory organization expected by the video hardware.

Here's how C language defines an MFDB structure:

**struct MFDB**

```
{
    char *fd_addr; /* address of raster */
    int fd_w; /* width in pixels */
    int fd_h; /* height in rows */
    int fd_wdwidth; /* width in words */
    int fd_stand; /* 0 for ST, 1 for standard */
    int fd_nplanes; /* how many planes */
    int fd_r1, fd_r2, fd_r3; /* reserved */
};
```

### Inside the MFDB

The first entry is a 32-bit (long word) *pointer* to an area of memory holding either the source image or destination area. A value of zero here is special: It tells the ST to use screen memory as the destination. You could use the actual screen memory address returned by *Physbase()*, but this technique is more portable (a value of zero on IBM GEM would point to the IBM display memory). Usually, this is a pointer to an integer or character array holding the screen memory image of the shape.

The next two entries, *fd\_w* and *fd\_h*, hold the width and height of the image in pixels. The next item, *fd\_wdwidth*, specifies the number of words needed to hold one row of the shape. You always round up to the nearest word; a width of 23 would round up to two words (32 bits), whereas a shape 16 bits wide would fit neatly in one word. This routine is most efficient when there are no fringe words outside the width of the image. For example, the shape with a width of 23 pixels has an unused fringe of 9 bits. This slows down the operation, since these unused bits have to be ignored during the screen merge. Of course, even if you use only 23 pixels, you have to pad out your data to a full two words for each row of the shape.

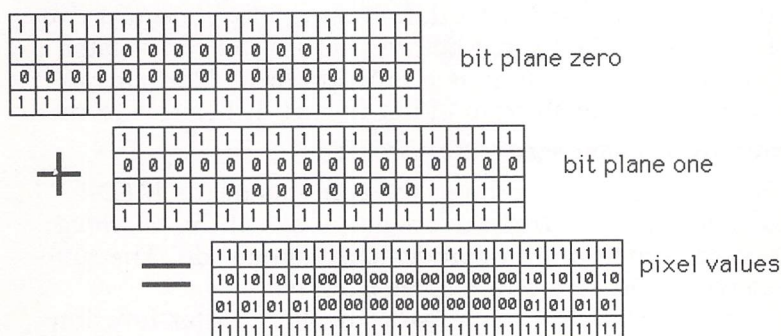
There are two standards for storing shape data, selected through *fd\_stand*. One format is compatible with other GEM-based computers, and the other is specific to the screen mem-

ory layout of the host computer (in this case, the ST). Our example programs use the standard color format, but you must use the system-specific format if you use this function in low-res ( $320 \times 200$ ) mode. The standard format has a value of 1, and 0 is used for the system-specific mode.

In monochrome mode, each word in screen memory holds the value of 16 pixels. Each bit in the word controls the corresponding bit on the screen, with the leftmost (most significant) bit controlling the first pixel on the screen. A value of 0 is white, and 1 is black (the opposite of most other machines in monochrome mode). Both the system-specific and standard modes use this same layout for monochrome mode.

In medium resolution, though, you need two bits per pixel to specify up to four colors per pixel (00, 01, 10, 11). GEM gets this information from two separate areas of memory. In standard mode, the left-bits of the pixel value come from one area of memory, and the right-bits from another. For example, refer to Figure 1 to see how the shape we use in Program 1 is composed.

**Figure 1. Bit Planes**



The ST-specific format for medium resolution puts each pair of words next to each other in memory. The left-bits come from the first word, and the right-bits from the second. The value of the first pixel on the screen is formed by pairing the leftmost bit of word 0 with the leftmost bit of word 1. This is illustrated in Figure 2.



### Figure 2. Word Pairs

one continuous section of memory

[illegible][illegible]

each pair of words merged for each line

We've already mentioned that you must use the ST-specific format to animate in low-res. Here we have four consecutive words that together define 16 pixels. The leftmost bit from each of the four words is concatenated to give a four-bit pixel value in the range 0-15.

## Image Conversion

If you use the standard form definition mode, you must convert it to ST-specific format before you use the copy raster function. Fortunately, there is a function provided that can convert back and forth from either format. It has this form:

```
vr_trnfm(work_handle,&srcfdb,&srcfdb):
```

Here, we are transforming the contents of the MFDB called *srcfdb*. The *vr\_trnfm()* function looks at the *fd\_stand* value to see what kind of conversion it should do. The function converts the data and toggles *fd\_stand*.

In deciphering the shape data, the copy raster function looks at the word width to figure out how many words to fetch per line, at the pixel width to see how many bits of the words should be used, and at the height to see when to stop drawing. The *fd\_planes* value tells the system how many planes of data there are (in standard mode) or how many consecutive words are needed to form 16 pixels.

You could use just one MFDB to control many shapes, but it's most convenient to use one MFDB structure for every shape, since different shapes have different widths and heights. In Program 1 we use two: one for the source image (a colored box) and one for the destination (the screen). When

you use zero for the *fd\_addr*, you can ignore all the other values in the structure.

### Using Vro\_cpyfm( )

Before you can do anything, you must define the MFDB structure within your program, as shown in Program 1.

The copy raster function looks like this:

```
vro_cpyfm(work_handle,mode,pxyarray,&shape,&destfdb);
```

You should be familiar with *work\_handle*, the same value needed by all the VDI functions, the value returned by the *v\_opnvwk( )* (open virtual workstation) call. *Pxyarray* holds the upper left and lower right coordinates of the source and destination rectangles. The copy raster function copies from  $(x1,y1)-(x2,y2)$  in the source image to  $(x3,y3)-(x4,y4)$  in the destination area. These coordinates are stored in this order in the array specified by *pxyarray*. The next two entries are the addresses of the MFDB structure for the source and destination rasters.

The *mode* value is one of the following values:

<b>XOR</b>	<b>6</b>
<b>REPLACE</b>	<b>3</b>
<b>ERASE</b>	<b>4</b>
<b>TRANS</b>	<b>7</b>
<b>REVTRANS</b>	<b>13</b>

This specifies the logical operation used to combine the source and destination bitmaps. REPLACE is the most straightforward: It completely overwrites the destination rectangle with the source image. Any 0-bits in the source image overwrite the background of the display. In ERASE mode, the areas of the screen overlapped by 1-bits in the display are erased (turned to zero). Areas covered by zeros in the source image are left untouched. In TRANSPARENT mode, only the 1-bits in the source image overwrite the destination image; areas overlapped by zeros are left alone. Reverse transparent mode inverts the ones and zeros in the source image before it merges the data as in transparent mode (the actual source data is unaffected).

The most interesting value, XOR, toggles the state of 1-bits in the display overlaid by 1-bits in the source image. You can use it to draw a shape on the screen; then reverse it



to remove the shape—without erasing the background in the process (we'll speak more about this magic later).

### Put( ) Your Shapes

In my programs I simplify all this with a function called *put()*, which is similar to Microsoft Advanced BASIC's PUT function. It assumes you've defined the destination MFDB for the screen as *destfdb* (see Program 1) and have initialized *destfdb.fd\_addr=0*. Along with the mode, you just pass to *put()* the address of the source image MFDB, followed by the *x* and *y* position where you'd like the shape to appear on the display. I've defined the four modes in my program as *fdb\_XOR*, *fdb\_REPLACE*, *fdb\_TRANS*, *fdb\_REVTRANS*, so you can use these constants instead of numeric values. A call to draw a shape at position (20,30) on the screen, using the REPLACE mode, might look like this:

```
put(&shape,20,30,fdb_REPLACE)
```

You can use two methods to animate an object without erasing the background graphics. The first method preserves and restores the background as the shape passes over it. Before you draw a shape, save the rectangular area that would be overlapped by the shape in a buffer. When you move the shape to the next position, you then restore the overwritten area from the buffer.

This works fine for one shape or for shapes that don't pass through each other. But imagine what happens when these kinds of shapes do pass over each other. Each shape first saves the image of the shape it overlaps. After the shapes pass through each other, they have both restored the area they overlapped, leaving behind images of the shapes.

The other method relies on a special binary mathematical operation known as *exclusive OR* (XOR). The binary truth table for XOR is (0 XOR 0=0, 0 XOR 1=1, 1 XOR 0=1, 1 XOR 1=0). If you know something about binary math, you can see that XOR works much like binary OR, or even like normal addition, except that when you XOR two ones together you get a zero. (Interestingly, binary addition yields the same result, but with a *carry* of one that must be added to the bit to the left.) When you copy a shape to the screen, you can specify the way the bits in the shape are combined with the bits in the background image.

### A Magic Stamp

Let's use a simple example. On a monochrome ST system, white is represented by 0 and black by 1 (the opposite of most computers—the ST monitor displays its screen in reverse to simplify programming). If you XOR a black shape (1) against a white background (0), you will see the shape  $0 \text{ XOR } 1$  as 1 (black). On the other hand, if screen memory is filled with 1's (black), and you attempt to XOR a shape made of 0's (white), you will see nothing, since  $0 \text{ XOR } 1$  is 1 (black).

But notice what happens if you put a black shape against a white background and then copy the black shape back on top of itself. The first operation is  $1 \text{ XOR } 0 = 1$ . When you XOR the black shape on top of itself, though, the operation is  $1 \text{ XOR } 1 = 0$ . The shape has removed itself. This method works no matter what the background data is; XOR is a reversible operation.

One way to think of XOR animation is that you're using a rubber stamp inked with a magic negative ink—an ink that reverses the color of whatever it touches. Naturally, stamping twice is the same as not stamping a shape at all. If you're careful, you can stamp two different shapes so that they overlap. Then, when you restamp those shapes, the background will be completely restored. The only problem is that the area where the shapes overlap is reversed. The 1's in the shapes XOR together in the overlapped area to give white.

It's a little more complicated with a color display, since the XOR is performed on the binary screen data. A binary pattern of 11 XORed with a binary pattern of 10 gives a result of 01. Two different colors, when overlapped, give a third color. Despite this color variation, though, using XOR is fast and effective as a technique for sprite simulation. When shapes are moving quickly, you rarely notice the strange overlap effects.

The XOR method of animation is fast and flexible. When we move an object, we first erase any previous image of the shape, then redraw it at the new position. (Notice that this is the opposite of moving the shape to the new position and then erasing the old image, since erasing the old image could remove part of the object at its new position. XOR lets us perform this erasure without cutting holes in the background display.



### Let's See Some Action

At this point, you're probably anxious to see all this really work. Type in Program 1, and compile and link it. It was written with the *Megamax C* compiler, but it should work with any other C compiler with the proper library and header files. When you run it, you'll see a rectangle smoothly moving and bouncing off the edges of the screen. Press the mouse button to exit the program.

The program is controlled by the *main()* function. We first define the variables we'll be using. The variables *x*, *y*, *prev\_x*, and *prev\_y* hold the horizontal and vertical position of the shape. The variables *xacc* and *yacc* store the acceleration of the shape, expressed in terms of displacements. *Xacc* and *yacc* are added to *x* and *y* each time to bump the shape to its next location. After we initialize the program, we used *init\_fdb* to create the source MFDB (*srcfdb*). The border off of which we'll bounce is defined by the width and height of the screen, adjusted by the width and height of the shape.

The values of *xacc* and *yacc* are set to 1, the slowest speed. Try different values here and recompile to see how fast you can get these to go.

Before we enter the main loop, the shape is *put()* at its original position. The animation loop first erases the previous image before updating the shape at its new location, so we have to give it a previous image to erase. (If we didn't, the first *put()* would draw an image instead of erasing, leaving behind an image of the rectangle).

Within the loop we save the current position of the shape, then adjust the position according to *xacc* and *yacc*. If these positions fall beyond the legitimate borders of the screen, the acceleration is reversed by negating it, and the *x* and *y* positions are adjusted back to their previous values.

### Vertical Synchronization

The *Vsync()* call makes sure the video beam is at the top of the screen before we erase the shape. This helps minimize flicker, since the shape is usually erased only within the first few lines of the display. We then draw the shape at its new position.

Since *Vsync()* usually returns only somewhere within 1/60 second later, it can be too slow if you want really fast

motion or if you want to animate many shapes. Delete it and recompile to see a real speed demon, albeit a flickery devil.

The loop continues until you click the mouse button, monitored by the *clicked()* function.

This program is expanded upon in Program 2, where we animate many shapes. In fact, you can change the constant NUMSPRITES to move as many shapes as you like. Of course, the more shapes you move, the slower they all move, but you may be surprised by how quickly they can go.

Program 2 is the core of an arcade-action game called "AstroPanic!," published in the premiere issue of *COMPUTE!'s Atari ST Disk & Magazine*, proving that arcade-action games are possible in C without resorting to machine language. The best results, of course, are achieved by bypassing the operating system and updating screen memory directly with machine language, but that's a challenge for more ambitious programmers. I think you'll find the method discussed here more than adequate for a wide variety of applications.



## Program 1. Shape Animation

```

/* Demonstrates animation with MFDBs */
#include <define.h>
#include <gemdefs.h>
#include <obdefs.h>
#include <osbind.h>
#include <stdio.h>

#define HIDE_MOUSE graf_mouse(M_OFF,&dummy)
#define SHOW_MOUSE graf_mouse(M_ON,&dummy)

/* global variables */
int dummy;
int work_handle, contrl[12], pxyarray[10];
int intin[128], intout[128], ptsin[128], ptout[128];
int work_in[11], work_out[57];

struct my_fdb
{
    char *fd_addr; /* address of raster */
    int fd_w; /* width in pixels */
    int fd_h; /* height in rows */
    int fd_wdwidth; /* width in words */
    int fd_stand; /* 0 for ST, 1 for standard */
    int fd_nplanes; /* how many planes */
    int fd_r1, fd_r2, fd_r3; /* reserved */
} srcfdb, destfdb;

#define fdb_XOR 6
#define fdb_REPLACE 3
#define fdb_ERASE 4

```

```

#define fdb_TRANS 7
#define fdb_REVTRANS 13

main( )
{
    int i,x,y,xacc,yacc,prev_x,prev_y,xborder,yborder;
    appl_init( );
    init_workstation( );
    HIDE_MOUSE;
    v_clrwk(work_handle);
    vswr_mode(work_handle,1);
    vst_color(work_handle,1);
    init_fdb( );
    xborder=work_out[0]-srcfdb.fd_w;
    yborder=work_out[1]-srcfdb.fd_h;
    xacc=1; yacc=1; x=0; y=0;
    put (&srcfdb,x,y,fdb_XOR);
    do
    {
        prev_x=x; prev_y=y; /* save old position */
        x+=xacc; y+=yacc; /* update to next position */
        /* check for bounce */
        if (x<0 || x>xborder) xacc=-xacc, x+=xacc;
        if (y<0 || y>yborder) yacc=-yacc, y+=yacc;
        Vsync( ); /* erase only near off-screen */
        put(&srcfdb,prev_x,prev_y,fdb_XOR); /* erase previous */
        put(&srcfdb,x,y,fdb_XOR); /* draw next */
    } while (!clicked());
    Terminate(0);
}

```



```

Terminate(flag)
int flag;
{
    SHOW_MOUSE;
    v_cslvkw(work_handle);
    appl_exit( );
    exit(flag);
}

/* waits for a period of time */
delay(period)
int period;
{
    evnt_timer(period,0);
}

/* returns TRUE if mouse button clicked, else FALSE */
int clicked( )
{
    int pstatus;
    vq_mouse(work_handle,&pstatus,&dummy,&dummy);
    return(pstatus != 0);
}

wait_for_click( )
{
    evnt_button(1,0x0001,0x0001,&dummy,&dummy,&dummy,&dummy);
}

put(shape,xpos,ypos,mode)
struct my_fdb *shape;
int xpos,ypos,mode;

```

```

{
    pxyarray[0]=0; pxyarray[1]=0;
    pxyarray[2]=shape->fd_w-1; pxyarray[3]=shape->fd_h-1;
    pxyarray[4]=xpos; pxyarray[5]=ypos;
    pxyarray[6]=xpos+pxyarray[2];
    pxyarray[7]=ypos+pxyarray[3];
    vro_cpyfm(work_handle,mode,pxyarray,shape,&destfdb);
}

init_workstation( )
{
    int i;
    work_handle=graf_handle(&dummys,&dummys,&dummys,&dummys);
    for (i=0;i<10;work_in[i++]=1); work_in[10]=2;
    v_opnvwk(work_in,&work_handle,work_out);
}

/* create a simple rectangular shape */
init_fdb( )
{
    static int image[] = {0xffff,0xf00f,0x0000,0xffff,
                          0xffff,0x0000,0xf00f,0xffff};
    destfdb.fd_addr=0; /* screen memory */
    srcfdb.fd_addr=(char *) image; /* raster memory */
    srcfdb.fd_w=16; /* width in pixels */
    srcfdb.fd_h=4; /* height in rows */
    srcfdb.fd_wdwidth=1; /* width in words */
    srcfdb.fd_stand=1; /* standard FDB */
    /* change number of planes to 1 for monochrome */
    srcfdb.fd_nplanes=2; /* two planes */
}

```



```

/* if you compile this with a monochrome system, remove next line */
vr_trnfm(work_handle,&srcfdb,&srcfdb);
}

```

## Program 2. Multiple Shape Animation

```

/* Demonstrates animation with multiple MFDBs */

#include <define.h>
#include <gemdefs.h>
#include <obdefs.h>
#include <osbind.h>
#include <stdio.h>

#define HIDE_MOUSE graf_mouse(M_OFF,&dummy)
#define SHOW_MOUSE graf_mouse(M_ON,&dummy)

/* global variables */

int dummy;
int work_handle,contrl[12],pxyarray[10];
int intin[128],intout[128],ptsin[128],ptsout[128];
int work_in[11],work_out[57];

struct my_fdb
{
    char *fd_addr; /* address of raster */
    int fd_w; /* width in pixels */
    int fd_h; /* height in rows */
    int fd_wdwidth; /* width in words */
    int fd_stand; /* 0 for ST, 1 for standard */
}

```

```

int fd_nplanes; /* how many planes */
int fd_r1, fd_r2, fd_r3; /* reserved */
} srcfdb, destfdb;

#define fdb_XOR 6
#define fdb_REPLACE 3
#define fdb_ERASE 4
#define fdb_TRANS 7
#define fdb_REVTRANS 13
#define NUMSPRITES 8

#define rnd(x) (Random( )%(x))

main( )
{
    int xborder, yborder, sprite;
    int prev_x, prev_y;
    int x[NUMSPRITES], y[NUMSPRITES], xacc[NUMSPRITES], yacc[NUMSPRITES];
    appl_init( );
    init_workstation( );
    HIDE_MOUSE;
    v_clrwk(work_handle);
    init_fdb( );
    xborder = work_out[0] - srcfdb.fd_w - 1;
    yborder = work_out[1] - srcfdb.fd_h - 1;
    for (sprite = 0; sprite < NUMSPRITES; sprite++)
    {
        xacc[sprite] = 0; yacc[sprite] = 0;
        while (xacc[sprite] == 0 || yacc[sprite] == 0)
        {
            xacc[sprite] = 4 - rnd(9);

```



```

        yacc[sprite]=4-rnd(9);
    }
    x[sprite]=rnd(xborder);
    y[sprite]=rnd(yborder);
    put (&srcfdb,x[sprite],y[sprite],fdb_XOR);
}
do
{
    for (sprite=0;sprite<NUMSPRITES;sprite++)

        prev_x=x[sprite]; prev_y=y[sprite];
        x[sprite] += yacc[sprite];
        y[sprite] += yacc[sprite];
        if (x[sprite]<1 || x[sprite]>xborder)
            yacc[sprite] = -xacc[sprite],x[sprite]=prev_x;
        if (y[sprite]<1 || y[sprite]>yborder)
            yacc[sprite] = -yacc[sprite],y[sprite]=prev_y;
        put (&srcfdb,prev_x,prev_y,fdb_XOR);
        put (&srcfdb,x[sprite],y[sprite],fdb_XOR);
    } while (!clicked( ));
    Terminate(0);
}
Terminate(flag)
int flag;
{
    SHOW_MOUSE;
    v_clswork(work_handle);
    appl_exit( );
}

```

```
        exit(flag);
    }
    /* waits for a period of time */
    delay(period)
    int period;
    {
        evnt_timer(period,0);
    }
    /* returns TRUE if mouse button clicked, else FALSE */
    int clicked()
    {
        int pstatus;
        vq_mouse(work_handle,&pstatus,&dummy,&dummy);
        return(pstatus != 0);
    }
    wait_for_click( )
    {
        evnt_button(1,0x0001,0x0001,&dummy,&dummy,&dummy,&dummy);
    }

    put(shape,xpos,ypos,mode)
    struct my_fdb *shape;
    int xpos,ypos,mode;
    {
        pxarray[0]=0; pxarray[1]=0;
        pxarray[2]=shape->fd_w-1; pxarray[3]=shape->fd_h-1;
        pxarray[4]=xpos; pxarray[5]=ypos;
        pxarray[6]=xpos+pxarray[2];
        pxarray[7]=ypos+pxarray[3];
    }
```



```

    vro_cpyfm(work_handle,mode,pxyarray,shape,&destfdb);
}

init_workstation( )
{
    int i;
    work_handle=graf_handle(&dummy,&dummy,&dummy,&dummy);
    for (i=0;i<10;work_in[i++]=1); work_in[10]=2;
    v_opnvwk(work_in,&work_handle,work_out);
}

/* create a simple rectangular shape */
init_fdb( )
{
    static int image[ ]={0xffff,0xf00f,0x0000,0x0fff,
                        0xffff,0x0000,0xf00f,0xffff};

    destfdb.fd_addr=0; /* screen memory */
    srcfdb.fd_addr=(char *) image; /* raster memory */
    srcfdb.fd_w=16; /* width in pixels */
    srcfdb.fd_h=4; /* height in rows */
    srcfdb.fd_wdwidth=1; /* width in words */
    srcfdb.fd_stand=1; /* standard FDB */
    /* change number of planes to 1 for monochrome */
    srcfdb.fd_nplanes=2; /* two planes */
    /* if you compile this with a monochrome system, remove next line */
    vr_trnfm(work_handle,&srcfdb,&srcfdb);
}

```

## CHAPTER SEVEN

# Pascal Programming





# A First Look at Pascal Programming

Tony Roberts

---

*When BASIC is not fast enough, or when you want to create real GEM applications, Pascal might be your solution. It's easier to create, maintain, and debug than machine language. But because it's a compiled language, it has greater speed than BASIC can provide. Here's a brief look at how the language developed and how it's structured.*

Judging by the notices left on bulletin boards and information services across the country, many Atari ST programmers who are looking for an alternative to BASIC and Logo are turning to Pascal in an effort to get the most out of their computers.

The Pascal alternative gives the faster execution that is possible with a compiled language, and, in addition, the language's structure supplies an environment that's helpful and comforting to many programmers. More importantly, Pascal can provide programmers with efficient and convenient access to GEM and its menus, windows, and dialog boxes.

BASIC is adequate for many types of programs, but when a programmer wishes to create a real GEM application, BASIC falls short. For such tasks, a programmer must turn to machine language, C, or Pascal. Of the latter two—both high-level languages—C produces faster, more efficient, and more compact code, but it's more difficult to use.

## **Pascal's Beginnings**

Pascal was born from a desire to give beginning programmers an ideal learning environment. At least, ideal according to Niklaus Wirth, Pascal's developer.

Wirth's philosophy is that programming is simply problem solving. His method for solving a problem is to break the problem down into its smallest parts and then to systematically solve those small problems. The expected outcome of this approach is that programmers who study Pascal will learn



good programming habits that can be carried forward no matter what language they eventually may use.

Pascal came into being in the late 1960s and early 1970s, years before personal computers were as ubiquitous as they are now. Programming students of that day did not sit at the keyboard learning by trial and error. They scratched out programs on paper, checking and rechecking to eliminate errors, before their programs were ever run.

When he created Pascal, Wirth hoped to produce a simple teaching tool. He had no expectation that the language would evolve into a successful development language. As a result, the original Pascal was quite without frills. However, though the original was sparse, Wirth wanted his language to be able to run on any computer, so he made the language extensible. That is, he made it possible to extend Pascal to accommodate the specific demands of different systems. As interest in the language grew over the years, this extensibility has made it possible for programmers to add the frills and functions needed to transform the teaching language into a language capable of producing sophisticated business and applications software.

Until recently, Pascal did not find much acceptance in microcomputer circles, largely because of memory limitations. The introduction of Borland's *Turbo Pascal* for IBM and CP/M computers, however, has given Pascal a boost. *Turbo* provided an easy-to-use programming environment that ran on the microcomputers most commonly found in businesses. Millions of programmers suddenly became aware of the power of Pascal and found they could write applications for their workplaces that ran faster than BASIC; at the same time programs were easier to create, debug, and maintain than machine language.

Any programmer who has worked with *Turbo Pascal*, or with any other implementation of the language, should have little trouble programming TOS applications on the Atari ST with a generic 68000 Pascal compiler. TOS applications, however, are unable to control the ST's GEM interface, and so they fail to satisfy the goal of the programmer who wants full control over the machine. To be truly useful to ST programmers, Pascal must be extended again, to provide easy access to menus, windows, and dialogs. These extensions require a sizeable addition to the original language and take time and study to master, but the time spent seeking out a compiler that con-

tains the extensions and learning to use them is well worth the effort.

So, although Pascal has grown up and changed a bit over the years, its underlying structure has stayed the same. Pascal remains a language of many rules, the theory being that if you program by the rules, you will have fewer unexpected problems.

Pascal is a strongly *typed* language, which means that variable use is rigidly controlled. An integer variable in Pascal cannot freely intermix with a real variable as is possible in BASIC. These strict guidelines help to reduce the possibility of errors when the program is run by screening out incompatibility when the program is created.

### Why Pascal?

As a compiled language, Pascal is a much faster language than BASIC, an interpreted language. As a high-level language, Pascal's syntax is more like English than are the low-level commands of machine language. A hybrid, Pascal offers speed approaching that of machine language without the need to communicate directly with the 68000 chip in its native language.

Another benefit of programming in Pascal is that, because it is high-level and structured, Pascal source code tends to be self-documenting. Because Pascal programmers generally use meaningful variable and procedure names such as *Phone\_Number*, *Total\_Amount*, or *Make\_Menu*, and because of the rules regarding how the program is to be written, it's usually much easier to understand a Pascal program listing than it is to understand a BASIC listing for the same program. Since Pascal programs make easier reading, they're also easier to modify and update as time goes by.

### When to Use Pascal

Pascal, unlike BASIC, really is not the correct tool for small, one-shot programming projects. Imagine dropping a few flakes of your breakfast cereal on the floor. You wouldn't rush out to the hardware store to rent a steam cleaner. You would probably get the broom and dustpan and be done with it. On the other hand, some jobs are too big for the broom.

Pascal is a good language for programs that will be re-used. Writing Program X in BASIC usually takes considerably less time than writing the same program in Pascal. However,



if you run the program often enough, the faster execution time and the convenience of the GEM interface of the Pascal program will make up for the difference.

The reason that Pascal programming is considerably slower than BASIC programming lies in the difference between an interpreted language and a compiled language. (Some Pascal interpreters exist, but their purpose is more to teach the language than to produce functional programs.) With an interpreted language, such as BASIC, the commands are interpreted and translated into machine language as the program is run. This translation takes its toll on execution time. In a compiled language, the translation occurs during the compilation phase. The finished program runs much more quickly, but creating the finished program is more complicated.

To write a BASIC program, you follow these general steps:

- Load BASIC.
- Write a few lines of code.
- Run the program.
- Make changes if necessary; then run it again.

On the other hand, here's what happens with a compiler:

- Load the editor.
- Write the program.
- Save the source code file.
- Load and run the compiler.
- If there are errors (and there will be errors), reload the editor and correct them.
- Reload and rerun the compiler.
- Repeat above two steps until all errors are corrected.
- Load and run the linker.
- After a successful link, load and run the program.
- If errors occur as the program runs, or if you want to make changes or modifications, reload the editor and repeat the sequence again.

As was mentioned earlier, the tradeoff for all this work is speed in execution. If you want to write an action game or a specialized database program or any application that manipulates large amounts of data, this speed is important. Pascal programming can be as enjoyable as any other programming, but, unless you have a serious application in mind, you may find it too time-consuming.

### How Pascal Works

In a broad sense, Pascal is much the same as BASIC. Program flow is controlled through various looping devices. The FOR-NEXT, IF-THEN-ELSE, and WHILE structures will be familiar to BASIC programmers. In both languages, variables are assigned values which are manipulated during the program run. The differences end, however, when it comes to the rules regarding how the program is written.

Although BASIC has some generally accepted conventions, it's really a free-form programming language; the style and structure of the program are up to you. For example, although the variable *i* is a commonly used loop control variable, there is no rule that requires it. Long lists of DATA statements usually appear at the end of a program, but it's really a matter of choice.

Pascal, on the other hand, has several rules that cannot be broken. Some programmers argue that these rules are impositions that hinder creativity; others feel that the rules provide a helpful frame in which to work. Time can be spent deciding how the program will *work* rather than deciding how the program will be *written*.

A Pascal program consists of the following items:

- Program identifier
- Constant declaration
- Type declaration
- Variable declaration
- Definition of functions and procedures
- The main program

The *program identifier* is a line that gives the program a name, as in the following line:

**program Stock\_Analysis;**

This line serves mainly to notify the compiler of the code's starting point. Note that all Pascal statements are terminated with a semicolon.

The program's *constant declaration* section supplies values that are not altered during the run of the program. Although it is not necessary to declare these values as constants, doing so makes the program easier to read and understand. Say, for example, that your program compares current weather conditions with the area's average conditions over the past 30 years. You might use the following constant declarations:



### CONST

```
Normal_Jan_Hi = 48.8;  
Normal_Feb_Hi = 51.4;  
Normal_Mar_Hi = 59.4;  
etc.
```

Later, in the body of your program, a statement might read:

```
if temperature > Normal_Jan_Hi then ...;
```

This line obviously is clearer than the statement would have been had the constant not been declared. Would a casual reader (or even you, the programmer) be able to decipher the meaning of this statement?

```
if temperature > 48.8 then ...;
```

*Type declaration* is a tricky subject that is usually ignored by beginning programmers. Pascal's predefined data types—which include integer, character, real, Boolean, byte, and, in many implementations, string—suffice for many applications. However, Pascal allows programmers to invent new variable types if they wish. For example, in a card game simulation, you might declare a variable type called *Suits* with the following statement:

### TYPE

```
Suits = (Club,Diamond,Heart,Spade);
```

This means that for a variable of the type *Suits*, there are only four possible values: Clubs, Diamonds, Hearts, and Spades. Assuming that the variable *card* had been declared as being of the type *Suits*, the following would be a valid program statement:

```
if (card = Heart) or (card = Diamond) then Write('It's a red card');
```

You may well write programs that require no type or constant declarations, but you'll be hard-pressed to write a useful program that makes no use of variables. Every variable in a Pascal program must be declared before it's used. This requirement not only helps keep a program organized and structured, but it tells the compiler what variables are coming and how the programmer plans to use them. Armed with this information, the compiler can look for and point out possible errors in variable use. Here's an example of a variable declaration:

### VAR

```
employee      : string[20];  
employee_id   : integer;  
Pay_Per_Hour  : real;  
Pay_Per_Week  : real;  
hours_worked  : real;  
full_time     : Boolean;
```

The first statement indicates that the program will use a variable called *employee*, which is a string with a length of 20 characters; *employee\_id* is an integer variable (a whole number). *Pay\_Per\_Hour*, *Pay\_Per\_Week*, and *hours\_worked* can all represent fractional amounts, so they are declared as real variables. And *full\_time* is Boolean, a variable that can have one of two values—True or False.

Now that the variables have been declared, the compiler can help you debug your program. How? Let's say the program contains these lines:

```
employee_id := hours_worked;  
full_timer  := True;
```

When the program is compiled, both lines will generate an error. In the first line, the variables are of different type. The compiler will not let you assign the value held in *hours\_worked* (a real value) to the variable *employee\_id* (an integer variable) because the types are incompatible. In the second line, the compiler would stop and warn that *full\_timer* is an undeclared variable. The programmer accidentally added an *r* to the variable name *full\_time*, and the compiler was unable to recognize it. This type of inconspicuous error in a BASIC program can be very difficult to track down and correct, but in Pascal, the compiler finds and points out the problem.

The colon-equal sign (*:=*) in the above lines is Pascal's way of *assigning* values to variables. The equal sign is used alone when *testing* values as in the following statement:

```
if full_time = False then Print_Check;
```

*Functions and procedures*, which are often called *subprograms*, are Pascal's workhorses. They are used to solve each of the small problems encountered in a program. They follow the same rules as programs. That is, they can contain their own constant, type, and variable declarations; they can contain functions and procedures of their own; and they must contain a main program section.



Subprograms allow programmers to extend the language. The Pascal extensions discussed earlier in this article are essentially collections of subprograms that you buy when you purchase a particular version of the language. Just as a specific set of extensions can make Pascal compatible with a specific type of computer, another specific set of extensions can make the language more compatible with a specific programmer.

If you're a mathematician, you might develop a set of subprograms that efficiently handle calculations common in your work. If you work extensively with data files, you'll probably develop a set of procedures and functions that open, close, and read files in a way that suits you.

If you wanted to print the output of your programs in Pig Latin, you could devise a subprogram to handle the conversion, and that subprogram could be included in any of your programs that required it.

The ability to reuse completed and debugged subprograms is a great timesaver for any serious programmer. Once you've built up a library of subroutines that solve the problems you most often encounter, you'll feel much more comfortable tackling larger and more sophisticated projects.

The *main* body of a Pascal program always comes at the end of a source code listing and is enclosed by the **begin** and **end.** keywords. The main program in Pascal is seldom long and generally serves to pass control to one or more subprograms. Assuming that the subprograms *Initialize*, *Menu\_Loop*, *Print\_Report*, and *Shut\_Down* had been previously declared and defined, the following could be a valid main program section:

```
BEGIN
    Initialize;
    Menu_Loop;
    Print_Report;
    Shut_Down;
end.
```

The *Initialize* procedure might make sure that certain variables are initialized and that the user is given any necessary instructions. *Menu\_Loop* would likely be where all the action takes place. Depending on the choice made by the user, the program could branch to one or several other subroutines. After *Menu\_Loop* is finished, presumably when the user selects the "Quit" option, *Print\_Report* takes over and prints out

some information based on what happened during the program run. Finally, the *Shut\_Down* routine takes over, making sure that disk files are closed and that the screen is returned to the proper color and resolution.

### Scope

Another difference between Pascal and BASIC is *scope*, which refers to the range in which a variable is effective. A variable that is declared at the top of a program is called a *global variable*, and its scope is the entire program. That is, the variable can be referred to or manipulated anywhere in the program.

When a variable is called within a subprogram, its scope is limited to that subprogram. When the subprogram is called, Pascal creates the variable and tracks it, but when the subprogram is exited, the variable is deleted and the space it occupied in memory is made available for other use.

The same variable can be reused in another subprogram or in the main program itself without conflict. This feature makes it possible to create subprogram libraries that can be reused easily in future programming projects.

### Learning the Language

If you've seen any of the numerous books on Pascal, you'll no doubt understand that the language involves a great deal more than these few pages can describe. If you want to learn the language, arm yourself with one or more tutorial texts and gather as many source code listings as you can find. The manuals supplied with most Pascal compilers don't attempt to teach the language. As you struggle to write your first programs, you'll look again and again to the references.

Find a compiler that provides a comfortable working environment. With some programs it's easy to switch from editor to compiler to linker to executing program. Some programs help you correct compile errors by reloading the source code and returning you to the problem with a helpful message. Some Pascal programs have been extended to take full advantage of the Atari ST and its GEM environment.

Don't be disheartened if the going is slow at first. Think back to your first days with BASIC, and you'll see that it, too, was tedious at the beginning. Once you've written a few programs and have developed an understanding of the language, you'll be guided more by your creativity than by the manuals.



# Event Management and Windows in Pascal

Program by Mark Rose  
Text by Bill Wilkinson

---

*Event management and windows in the GEM environment are two of the most misunderstood topics of ST programming. This article sheds some light on these areas and shows you what a well-structured GEM program looks like.*

When Apple introduced the Macintosh computer, the company dubbed it "the computer for the rest of us." Since a fully equipped Macintosh originally cost as much as a pretty fair used car, Apple certainly wasn't referring to the price. Instead, the emphasis was that the Macintosh is a powerful machine, easy for even a complete novice to use. So true. Imagine being able to copy a file by pointing to a little picture and then dragging an outline of that picture to another picture which shows a second disk drive. No more cryptic commands such as

**PIP/QVF B: \WORK \=A: \BUSY \\*.DAT**

or worse. Apple's question: What could be better?

Atari's response: How about a computer that is affordable for the "rest of us"? Seen at first glance, the desktop of the Atari ST looks amazingly like that of the Macintosh. Upon closer examination, you see several not-so-subtle differences. Just as an example, the Macintosh allows custom pictures (icons) to be drawn and associated with particular types of files. The Atari ST has only three fixed icons—folders, programs, and data files—which are often more than a little confusing because, for example, BASIC programs are considered to be data files.

Still, the most outstanding features of the Macintosh are maintained: a mouse that is easy to use, a visual desktop, and, naturally, windows. The logical consequence of all this is that programmers who want their products to mesh well with the

ease and grace of the ST must learn how to “do windows.”

This discussion then will focus on two of the most misunderstood topics of ST programming: event management and windows in the GEM environment. The program accompanying this article is designed to show a well-structured GEM program. Since we are principals in Optimized Systems Software (OSS), and since OSS produces a popular language for the Atari ST known as *Personal Pascal*, it is not surprising that this program is written in *Personal Pascal*.

You say you don't have *Personal Pascal*? No problem. This isn't a practical program anyway (though it certainly could serve as the basis for one). Its purpose is simply to instruct you in programming techniques, and the methods and algorithms apply equally well, whether you program in Pascal, C, Modula-2, assembler, or any language which gives you full access to the power of GEM. (Which is another way of saying that BASIC and Logo users don't really need to know much of this stuff. But when you get tired of the self-imposed limitations of those languages, come back and read this again.)

*Personal Pascal* comes with a special library of functions and procedures that give programmers access to a reasonably good selection of GEM features. In reading the example program and the following material, you will find that several of these special library routines are not explained. This is only reasonable; who cannot deduce the purpose of a procedure named `Paint_Color`? On the other hand, we'll briefly explain routines with less obvious names and/or purposes. Long descriptions are unnecessary, since *Personal Pascal* users can find them in their manuals, and users of other languages will either have to invent their own routines or use names and calling sequences dissimilar to those given here. We have attempted to make the program as readable as possible so that it will not cause any confusion.

### What Window\_Demo Does

In good programming style, we've given our program a readable name: “`Window_Demo`.” In typical computer style, when we put it on disk, we were forced to call the program “`WINDDEMO`”, since TOS allows only eight characters in a filename. After you have compiled the source code (or otherwise made a copy of the runnable program), you simply need to click on the icon in the desktop which bears the name



"WINDDEMO.PRG". It will load and start to run.

The first obvious change will be in the menu bar at the top of the screen. Along with the familiar Desk designation, you will find Sizes, Shapes, and Patterns. If you move the mouse pointer up to the menu bar, you can touch on one of these names and be rewarded with a drop-down menu. For example, if you choose Sizes, a drop-down menu will appear containing the selections Small, Medium, and Large. Beside one of these selections will be a checkmark. If you choose a different selection and click on it, the drop-down menu will disappear, but when you reselect it, your new choice will have the checkmark next to it.

Similarly, there are three shapes (Square, Circle, and Wedge) and three patterns (Solid, Checkered, and Open); you can choose one shape and one pattern. The real magic of this program occurs, however, when you point the mouse at the Desk title in the menu bar: In addition to any desk accessories already present on your boot disk, a selection titled New Window appears. Choose it by clicking on it, and the action will begin.

First, a window appears, filling the desktop except for the menu bar area. Its title is Window 1 (or a higher number if this is the second or subsequent time you have chosen New Window). This window is typical of the characteristics of GEM windows in several ways: it has a title, move bar, close box, and size box. The implication is that you can point the mouse at the size box and drag the corner of the window to make it smaller. You can also move the window by dragging it via the bar containing its title, though we suggest you do this *after* you have shrunk it somewhat. If you can run the "WINDDEMO" program at this time, we suggest that you create several windows right away, shrinking and moving each to different sizes and positions on the screen.

Now, the meaning of Sizes, Shapes, and Patterns becomes obvious: If you point the mouse somewhere in the interior of the frontmost window, a shape of the type you "check-marked" will appear. Its size and interior pattern will also match your choices. If you have a color monitor, the program chooses three colors cyclically (more on this later). You can move the mouse pointer and click to request as many as ten shapes per window.

Notice that you can change your choice of shape, size,

and pattern at any time, either while working in the same window or before moving to another window. The best part, though, comes when you move or resize your windows again: All of your work is not forgotten. The objects remain in the same relative positions within the windows.

If you ran this program in a nonwindowed environment (for example, on a typical eight-bit computer like an Atari 130XE), none of this would be very difficult or extraordinary. But let's take a closer look at the programming techniques necessary to accomplish all this in GEM's multiwindow environment.

### When the Bottom Is the Top

We'll follow the logical flow of the program, not the physical order of the listing. With Pascal programs, that usually means we have to start at the bottom of the listing, because the top of the program starts there.

Before we start, let's introduce some of the **types** and **variables** used in the program. In particular, you need to know that any **types** which seem undefined in the program are undoubtedly described in the file "GEMTYPE.PAS", which is included in this compilation via the following program line:

```
{ $I GEMTYPE.PAS }
```

The \$I is also used to include the files "GEMCONST.PAS" (predefined constants) and "GEMSUBS.PAS" (external support routines). This mechanism, or similar ones, is common to many compiled languages. Some languages, such as C, have a feature like this in the definition of the language. Others, such as Pascal, have acquired one through common usage: *Personal Pascal's* use of \$I has historical precedents.

In this program the most difficult **type** to understand is probably `window_info`, the last **type** defined. If you remember what appeared on the screen when we ran this program, you will soon see why this **type** is so complex: Each window has a title (name) and may contain from zero to ten objects (`obj_count`). The objects must be described, and for this we use an `object_list`.

In fact, this list is an array of ten elements, each of which is called an `object_info` and each of which describes the shape, location (both horizontal and vertical within the window), size, pattern, and color of a corresponding object. With a little perusal, you will see the rationale for the names and



organization of the given **types**. What is perhaps not so obvious is the need for all this record keeping. Doesn't GEM take care of windows for us? As we shall see, the answer is an often surprising no.

Before we leave the **types**, we want to mention one nicety of Pascal in general and *Personal Pascal* in particular: Notice the definition of `shape_kinds`. Nowhere else in this program will you see `square`, `circle`, or `wedge` defined as either **var** or **type**. Simply by defining `shape_kinds` as we have, we have produced an *enumerated type*. As a consequence, at any place in this program where we need a *value* to assign to a variable of type `shape_kinds`, we can use *only* one of these three names as that value. Although obviously represented internally to the compiler as numbers, enumerated types allow us to write exceptionally well-structured programs, since it's impossible to make the error of assigning a number too small or too large to a variable of such a type.

On, then, to the end of the program, which is actually its beginning. Look for the comment only a dozen lines or so from the end, which marks the start of the main program code. Can you believe how small this program really is? If we were to translate the program lines into English, this is how they might sound:

If we can successfully initialize GEM, then we need to initialize our window descriptors, describe ("make") our new menu bar, draw and adjust that menu, and make sure the mouse is represented (on the screen) by an arrow. Only then can we call for our master event scheduler (which really does all the work). When we are finished processing events, we will erase the menu that we drew and exit cleanly from GEM.

Seems so simple, right? But then what are all these pages and pages of listing for? Hang in there.

**A special note:** In writing this program, we purposely followed some OSS programming standards. Most of them are for form only, but at least one of these rules will make it easier for you to understand the program: Library and built-in routines always start with a capital letter; routines in the current program always start with a lowercase letter. Thus, you can instantly tell that `Init_Gem` and `Draw_Menu` among others are library routines, while `init_windows` and `make_menu`, for example, will be found somewhere in this program listing.

The purposes of the library routines used in the main program control all seem fairly obvious: `Init_Gem` simply asks GEM to initialize itself. `Draw_Menu` causes the menu pointer passed as its argument to replace the current contents of the menu bar. `Set_Mouse` changes the form of the mouse icon (in our example, it changes to the usual arrow, in case the busy-bee icon is present because the program was loaded off disk). And so forth. Once again, we'll not attempt to explain every one of these library routines. If you use a little imagination, you'll probably guess correctly what they do.

From here on, the subtitles used in this narrative will correspond to the names of the program routines being described. Just scan through the listing until you find a **procedure** or **function** of the same name and prepare to learn more about GEM windows.

### **init\_windows** (page 308)

This simple routine initializes our *windows* array. Setting the *handle* to indicate `No_Window` tells later routines that this element of the array is available for use. Note that the name we build for this array element corresponds to its array index. Neat and clean. Note also another powerful feature of Pascal: The **with** construction allows us to omit the record name designator in assignments and expressions. If we had not used a **with** here, both *handle* and *name* would have needed the record designator *windows[index]*. preceding them.

### **make\_menu** (page 321)

This routine illustrates one of the nicest features of a flexible GEM library: the capability of creating menus dynamically. There is no reason that you could not create routines to do this in any language, but as of this writing no standard library routines to do so exist for other than *Personal Pascal* users.

The concept is fairly simple: Ask for a new menu, telling the library routine how many titles and subtitles it will contain and what name to "register" it under (the name which appears under the standard word *Desk* at the left end of the menu bar). Then add a bunch of titles, each associated with the menu *handle* returned by `New_Menu`. Finally, add subtitles (or **Menu Items**), associating each with the menu handle and the appropriate title. And that's about it.



In our `make_menu` routine, we also set up the current (default) values for the size, shape, and pattern of objects which will be drawn.

### **adjust\_menu** (page 316)

Another simple routine: After any change to the menu (or after it's drawn for the first time), we need to put checkmarks on the menu items chosen by the user. So, for each of the three possible checked items, we call `Menu_Check`, telling it the handle of our menu and the item index which will get the checkmark. The last parameter is always a Boolean true or false, which indicates whether to add or remove the check.

### **event\_loop** (page 320)

It's been far too easy so far, hasn't it? Here's where the going gets tough. The real basis for this entire program is the simple **repeat-until** loop of this procedure. Yet, really, only three things seem to happen here.

- We get an event, which can be either a message or the press of a mouse button (`E_Message` or `E_Button`).
- We disallow further events until we are done with menu processing via the call to `Begin_Update`.
- If the event is a message, we do one thing (`do_message`).
- If it is not a message, we assume it is a button click and do something else (`do_button`).
- We allow more events to occur (`End_Update`).

The `Get_Event` library function (which is essentially identical to `evnt_multi` in the Atari C library) deserves some attention. The first argument to this function tells GEM what kinds of events our program will handle. Each bit of the argument specifies a class of events that we are looking for. In this instance, we've chosen only the message and button events mentioned above.

The nature and number of the rest of the arguments to this function depend on which events we have chosen to monitor. Generally, those which we have coded false, junk, or zero do not enter into this particular usage. Of the other arguments, the three following the first one indicate what kind of button event we are waiting for (in this case, for the left button to be pressed). The `msg` buffer is actually a general-purpose array which is used to return all sorts of information to us (as

we'll see later). Finally, `mouse_x` and `mouse_y` return the horizontal and vertical position of the mouse (in this case, when the button is pressed).

Simple? Perhaps, but consider that a more complex program would have to handle keyboard events, mouse and window "collisions," and timers that count down to zero. And you really *should* handle all these events, because GEM has a few nasty bugs which crop up when a program ignores one or more of them, usually when desk accessories are also active. For our example, however, let's see where these events can lead.

### **do\_message** (page 318)

If you've thought about how this program is used, you might be wondering just how much work the programmer must do. For some operations, there is surprisingly little to do. For example, if the user moves the window via the move bar or changes its dimensions via the size box, the first entry in the *msg* buffer indicates this choice. Other elements in the array give the handle of the window in question as well as its new location and size.

If you examine the code of the `do_message` procedure, you'll see that whenever we get either a `WM_Sized` or `WM_Moved` message (WM stands for Window Message), we simply pass the request back to GEM as a `Set_WSize` call. This may *not* be what you want to do in your own program. For example, we place no lower limit on the size of a window; the user can shrink it down to a box with no active drawing area. Is there a certain minimum size you want your program to support? Then just test for it here and ignore messages which try to make the window too small (or, alternatively, translate a request for a too-small window into a `Set_WSize` to your acceptable minimum).

But before we can go moving or changing the size of any window, we have to create one. Do you recall what causes this to happen in this program? The user must click on the New Window item under the Desk title in the menu. But *all* menu selections cause a message of `MN_Selected` (MeNu Selected), so let's see what happens when we call the `do_menu` routine. First, note that we are passing two elements of the message buffer array as parameters to the procedure. When the message is `MN_Selected`, these elements contain the indices to the



selected title and item (subtitle) within the menu. Is it beginning to make sense?

### **do\_menu** (page 317)

Since there are only four possible titles in our menu, we need to check for only these cases. But we've never found out the title index for the system-supplied Desk title. Not to fear: If it's not one of the other titles, it *must* be the Desk title, and that further means it must be our New Window item. Do you see why? It is the only item under that title which "belongs" to our program—any other items belong to desk accessories.

If the user has chosen to change the current shape, size, or pattern for subsequently selected objects, we simply erase the checkmark from the prior "current" item and make note of the new current item. The call to `adjust_menu` near the end of this procedure is used to place a checkmark beside the new selection. And the last line, a call to `Menu_Normal`, is needed to "deselect" the title which has been placed in reverse video by GEM. This is only cleanup work, but it's necessary.

The real work here occurs when the user has indeed requested a New Window by clicking on that item under the Desk title. But, in keeping with good structured programming techniques, that will be handled by yet another routine.

### **do\_open** (page 315)

If we get here, it's time to open a new window. Since GEM limits us to seven windows, we first check to see whether that many already exist. If this would be the eighth window, we issue a nasty message in the form of an alert box. Notice how simple alert boxes are in GEM: a single call to `Do_Alert` passing a single string and a default button number. The default button number is always necessary, but it may be zero. When multiple buttons are present, the value returned from `Do_Alert` indicates which button has been "pressed" and is not just junk, as it is here.

Now, finally, we're ready to open a new window. First, we call `which_window` (which will not be explained since its usage and coding are obvious) to find an empty slot in our `windows` array. Again, we get to use Pascal's **with** keyword to simplify the coding.

Notice how easy it is to ask GEM for a `New_Window`.

The first parameter may look confusing, but it's simply a word wherein individual bits request various possible features of a window. Here, we are asking four things:

- To give the window a name
- To allow the user to change the size of the window
- To allow the window to be moved via the move bar
- To allow the window to be closed

Several other options are also possible—for example, an info (or subtitle) line, a full box, and scroll bars. (A good example of a truly full-blown GEM window is that which appears in the desktop to show you the contents of a disk directory.)

If we have asked to give the window a name (and we did), we must pass the desired name as the second parameter. The last four parameters specify the *limits* of the position ( $x$  and  $y$ , horizontal and vertical) and *size* (width and height) of the window to be created. A special feature of the *Personal Pascal* library is that if all four of these parameters are zero (as they are here), a window of maximum size placed anywhere on the screen is allowed. This is convenient, since the program need not worry about screen resolution. Theoretically, GEM can still reject our request, so we provide another alert box message to handle this possibility.

If we finally get through all that, we can open our window, using the handle returned by the call to `New_Window`. The trailing four parameters to `Open_Window` are, once again, the position and size of this window. This time, though, they specify how the window will appear when it is first drawn on the screen. The meaning of all zero parameters is the same: Make the window as big as possible. Note that, even if we created a smaller window here, the zeros in the call to `New_Window` would allow the user to expand the window to full screen size.

And that's it; the rest is clean-up work. We can't emphasize this enough: *You should not draw anything in the window at this time, even if you know what is going to appear there.* Doing so is one of the most common mistakes made by newcomers to GEM. Don't do it. Wait for GEM to *tell* you when it wants you to draw your goodies in the window. Confused? Keep reading.



### **do\_button** (page 319)

In discussing the event loop, we mentioned that we were asking GEM for both messages and button events. We're not done with messages yet, obviously, but let's take a peek at what the button events are for. If you look again at the code for `event_loop`, you'll notice that `do_button` is called with two parameters: the horizontal and vertical position of the mouse when the button is pressed. What then?

Not too surprisingly, the first thing we do here in `do_button` is wait for the user to release the button. This involves another call to `Get_Event` (which we will not detail, but you should note the differences between this call and the one in `event_loop`). The rest of the routine is simple: *If* the mouse button is pressed while the mouse pointer is within the front window and *if* we "own" the front window, then we want to add a new object to that window at the given mouse position.

But we have an artificial restriction that no window may contain more than ten objects. If the current window already contains that many objects, we'll issue another one of those handy alert boxes. However, if everything is okay, we'll add this object to the current window.

### **add\_object** (page 313)

This is by no means the most elegant of routines. In fact it looks downright messy. But it performs the very important task of adding an object's definition to the `object_list` within the appropriate element of the `windows` array.

After adding this object to the count, we obtain the coordinates (and size) of the current window. This is necessary because the mouse coordinates are given in absolute units, yet we want to store them in the list using numbers which are relative to the origin of the current window. Confused about where *x* and *y* are defined? Once again, notice the handy **with** clause: They're part of the `object_info` within the `object_list`, which in turn is part of a `window_info` record in the `windows` array (whew!).

The size, shape, and pattern are set within this same `object_info` entry using values which will be meaningful later. The color is chosen to cycle through the three possible foreground colors of a medium-resolution screen. (Though the

color choice is unnecessary in monochrome and uses only 3 of 15 available colors in low resolution, this method at least insures compatibility of the program with all resolutions.)

We now need to draw the object we just added. Since the rest of this routine (from `Set_Clip` onward) is similar to what happens during a redraw, we'll postpone discussion of the called routines for a few more paragraphs— especially since we shouldn't even be adding objects yet. We've not learned how to clear out the interior of our window yet.

### **do\_message** (page 318)

Back here again. In the previous discussion of this routine, we purposely left `do_message` before finishing it. We didn't look at what happens when some of the possible messages occur. For example, when a user asks to close a window (by clicking on the close box at the top left of that window), we call the `do_close` routine with the handle of the window that's to be closed. Even though `do_close` is one of this program's procedures, we don't need to discuss it here since it's very simple.

The `WM_Topped` message does need some explanation: When the user clicks on a window which is *not* the current (front) window, GEM tells us to prepare to make the selected window become the front one. If we have no clean-up work to do, we need do no more than call the library `Bring_To_Front` routine, *even if that window is partially obscured by another window*. Once again, this concept is misunderstood by newcomers to GEM, who try to redraw the contents of that window at this point. Don't do it. Wait for a `WM_Redraw` message.

A `WM_Redraw` message is the real heart of the GEM windowing system. Any time GEM determines that all or part of one of your windows needs to be redrawn, it will tell you so with this message. This is why you shouldn't start drawing in a window when you first create it, or when the user moves it or brings it to the front: GEM will *tell* you when it is time to do so. As a point of interest, you usually don't get in big trouble if you violate this rule unless your program is a desk accessory or unless there are desk accessories also doing their own thing with the screen. But why take chances?

Okay, so what do we do when we get a `WM_Redraw` message? We call `do_redraw`, passing the window handle and the position and size as parameters.



### **do\_redraw** (page 312)

Now we come to the most important point of this entire discussion: What must your program do when GEM hands you a redraw message? Consider, as an example, the situation where several overlapping windows are covered by a desk accessory's window. What happens when the desk accessory finally asks GEM to remove its window from the screen? Each of those several windows must be redrawn. And we're about to show how.

A simplistic approach might be to start with the rearmost window, completely drawing its contents, and then work our way forward. But, if several complex screens need updating, even the powerful 68000 used in the Atari ST is a little short on speed for this method. GEM knows this, so it always tries to keep the amount of screen processing to a minimum. It does this by keeping track of the screen's contents through a set of rectangles. Before we get into how that relates to our program, let's look at the housekeeping which is necessary in any screen redraw routine.

GEM has given us the handle of the window it wants us to work with, and that handle is what our `do_message` routine has passed to this procedure. So we need only use `which_window` to determine which element of our *windows* array contains the information describing the window in question. Before doing anything else, we temporarily remove the mouse from the screen. (Sidelight: Many early ST demo programs omitted this last step. That's why many otherwise nice screen displays had "holes" in their middles when the mouse got moved later.)

The call to `First_Rect` is our first evidence of the way GEM keeps track of all the windows on the screen. This particular call asks GEM to tell us the position (*x* and *y*) and size (*w* and *h*) of the first rectangle which needs to be redrawn. Note the **while** loop which terminates only when GEM uses a size of zero to indicate that there are no more such rectangles.

GEM's redraw rectangles may or may not coincide with the boundaries of our window, so we use the call to `Rect_Intersect` to limit that position and size only to the portion that lies within the window in question. `Rect_Intersect` is a Boolean (true or false) function which obligingly tells us whether any portion at all of GEM's redraw rectangle lies within our window. Presuming that at least some portion of it does come

within our bounds, we're ready (at last) to start modifying the screen.

We begin by telling GEM to limit its drawing of objects (which can include text) to the now-intersected rectangle in question. Then we use `Paint_Rect` to change that entire area to a solid white color. Finally, we call yet another routine to draw the list of objects that the user has placed in this window.

This last call, to `draw_list`, is really the only part which must be modified if we are using this program to update some other kinds of windows (such as text windows). All other calls in this procedure can and should remain essentially unchanged.

We'll look at how the list of objects is drawn in a moment, but first let's look at the tail of this routine. Note the call to `Next_Rect`, where we get another possible rectangle which needs updating. Remember, if the rectangle size is zero, the **while** will stop this whole process. At that time, we'll allow the mouse to reappear. This `do_redraw` procedure, or one very similar to it, should be the heart of any proper window-oriented GEM program.

### **draw\_list** (page 311)

This routine and the next are the only ones which are truly unique to our particular program. And `draw_list` is extraordinarily simple: It obtains the coordinates of the upper left corner of the working area of the current window (via the call to `Work_Rect`; the size is also obtained, but we ignore it here). Then, one by one, it processes the list of objects (if any) in this window.

If you have trouble following some of the code here, remember the importance of **with**. The reference to `obj_count` is actually a reference to `windows[index].obj_count`; objects and handle are similarly referenced).

### **draw\_object** (page 310)

Now we're finally going to call a routine to put a shape on the screen. After all the build-up, this routine is almost anticlimactic. We tell GEM about the user's choice of pattern and which color to use, and then a shape of the user's choice is drawn in the selected size. And that's the end.

Perhaps we should explain the seemingly strange numbers used in the various "Paint" calls. For example, a call to



`Paint_Rect` does what the name implies; it paints a rectangle. In keeping with common GEM usage, we specify a rectangle by its top left coordinates and its width and height (and since the width and height are the same, we should get a square). But as smart as GEM is, it still hasn't figured out that we're trying to draw within the bounds of one particular window, so we must adjust the *x* and *y* position of the square by adding the coordinates of the top left corner of our window.

Similarly, the `Oval` (circle) and `Arc` (wedge) must have their positions adjusted and use an appropriate radius value (halved in the case of the circle to make it appear the same size as a square).

### Six Exercises

If you managed to wade through this discussion, you're probably ready to tackle a window-based GEM program. May we suggest a few projects or areas for further study.

- 1 The most obvious project would be a translation of our program from *Personal Pascal* to another language (or even to another Pascal dialect if the Pascal you are using doesn't have similar GEM support libraries). This task shouldn't be very difficult; you need only to figure out the routines in AES and VDI that the *Personal Pascal* routines are calling. Generally, our library routines consist of either a one-for-one translation of a lower-level routine or a set of logical calls to several such routines. On a few occasions, we've actually created more routines in order to avoid forcing the user to pass a parameter which selects between two possible meanings of a VDI or AES routine.

- 2 Another project that interests us is an improvement to the `draw_object` procedure. Currently, we always draw each object in the object list. But suppose that one or more objects are not even partially within the bounds of our clipping rectangle (see `do_redraw`): Why should we take time to call GEM to redraw hidden objects? We wrote this program for clarity and simplicity, and GEM doesn't care because it displays nothing when an object (or part of an object) is drawn outside the clipping rectangle. It wouldn't be very difficult to use `Rect_Intersect` to find out whether the square or circle is within the clipping area. The wedge is somewhat harder, but a little work with geometry will help a lot here.

3 An easy add-on might be to allow the user to choose the color of an object (another menu item?). Doing this right implies that your program must determine the resolution of the current screen (not too hard—a call to an XBIOS routine). It would be nice to display the color choices by color instead of by name, but that part of this project is probably beyond all but expert GEM programmers. Stick with allowing the user to choose orangish purple and the like.

4 A severe shortcoming of our program is obvious if you run it in medium (four-color) resolution. The so-called circles become tall, skinny ovals, and the squares are anything but. What has happened? Quite simply, we chose to ignore the fact that the color pixels are not truly square. In low-resolution mode, the discrepancy is almost unnoticeable, but in medium resolution, the unbalanced “aspect ratio” becomes apparent. So a really good program should determine the aspect ratio of the screen currently in use and compensate for it when the objects are drawn. This method won’t be completely accurate, but for starters you could try doubling the width of drawn objects in medium resolution. Or get fancy and calculate actual multipliers; then your program should work on virtually any present or future GEM system.

5 Consider what a multiwindow text-editing program must do. Everything we’ve said about windows and redraw events applies equally well here. Drawing text on the screen can be painfully slow if you don’t carefully choose your window locations and redraw areas. You’ll need to do a bit of homework or experimentation before tackling this one if you want fast screen updates.

6 Our restriction of a maximum of ten objects per window is purely artificial. In fact, simply by changing the value of the constant `max_objects` at the head of this program, you can raise that number dramatically. Still, any constant number here is actually a kind of arbitrary limitation. A better method would be to use a linked list of objects, where each new object’s definition would be dynamically allocated via Pascal’s **new** command. And, of course, more flexibility in the definitions of the objects would be nice. We have the beginnings of a real drawing program here. What about lines, polygons, fill patterns, and so forth? If you’re really ambitious, our simple example could turn into a winner of a program.



## Window\_Demo

```
program Window_Demo;
```

```
const
```

```
{ We're going to give names to a few special numbers here so we can use the
  names from now on, instead of just pulling numbers out of a hat. }
```

```
{ $I GEMCONST.PAS } { Constants from the Pascal GEM libraries }
max_windows = 7;    { The maximum number of windows we can have open }
max_objects = 10;   { The maximum number of objects drawn in each window }
```

```
type
```

```
{ $I GEMTYPE.PAS } { Type declarations from the Pascal GEM libraries }
```

```
{ We need to record several pieces of information about each object so we
  can properly draw it within a window. We'll hold this information in a
  record of type 'object_info'. }
shape_kinds = ( square, circle, wedge );
object_info = record
```

```
    shape: shape_kinds;
    { The 'x' and 'y' fields hold the position of the object
      "relative" to the upper left corner of the window's
      workspace area. }
    x, y, size, pattern, color: integer;
end;
```

```
{ Since each window can hold several objects, we need some suitable data
  structure to hold information on multiple objects. For simplicity in
  this demo, we have chosen a simple array. For more complex applications,
  a linked list structure would probably be more appropriate. }
obj_range = 0..max_objects;
```

```
object_list = array[ 1..max_objects ] of object_info;

{ Now we come to the definition of the record which will hold all the
  information about a window. We need to hold the "handle" returned by
  GEM, the name of the window, and the objects that were drawn in it. }
window_range = No_Window..max_windows;
window_info = record
    handle: integer; { GEM's window handle }
    name: Window_Title; { Name of our window }
    obj_count: obj_range; { Count of objects in the window }
    objects: object_list; { And the array that holds them }
end;

var
{ Now we get to our global variable definitions. We really only need a few
  pieces of information globally. }

{ Variables needed to draw or keep track of our menu: }

{ The pointer used to refer to the entire menu for drawing or erasing }
menu: Menu_Ptr;

{ Menu title indices }
size_title, shape_title, pattern_title: integer;

{ menu subtitle indices }
size_small, size_medium, size_large,
shape_square, shape_circle, shape_wedge,
pattern_solid, pattern_checked, pattern_open: integer;

{ Variables used to hold information about the windows we have open: }
```



```
{ The array that holds all of the window information records }
windows: array[ 1..max_windows ] of window_info;
{ and the count of how many windows we have open }
window_count: window_range;
```

{ We also need to keep track of the current choices the user has made about the next object to be drawn. These choices are stored as the menu item indices which the user selected. When a new object is added, these values will be converted to the values actually stored in every 'object\_info' record. }

```
    cur_size,
    cur_shape,
    cur_pattern: integer;
```

```
{ $I GEMSUBS.PAS } { Subprogram definitions from the Pascal library }
```

{ init\_windows - This routine performs all the initialization of the window information variables. We need to set the count of windows to zero, since we don't have any windows open yet. We also set the 'handle' field of each record to the value 'No\_Window', an illegal window handle value defined in the GEMCONST.PAS file, as an indication that the record is free to be used when a new window is opened. We will also set up the name of each window here. In a real program, you would probably want to allow the user to specify a name for a window, so you'd want to set up the name sometime later, perhaps when a window is actually opened. }

```
procedure init_windows;
```

```
var
    index: window_range;
```

```
begin
  window_count := 0; { We start out with no windows. }
  for index := 1 to max_windows do
    with windows[index] do
      begin
        handle := No_Window;
        { Each window name is ' Window # ', where the '#' is replaced by
          the index of the window in the 'windows' array. }
        name := concat( ' Window ', chr(ord('0')+index), ' ' )
      end
    end;
  end;

  { which_window - At various times we will need to convert from a window
    handle which was returned by GEM into the index of the window in our
    'windows' array. The following routine searches the array for a window
    whose handle matches, and returns the index into 'windows' if a matching
    window is found. Otherwise, the value 'No_Window' is returned to show
    that no matching window was found. This routine can also be called with
    the value 'No_Window' as the 'wind_handle' parameter. In this case it
    will look for a record in 'windows' with 'No_Window' as a handle (i.e.,
    a free entry), so we can also use this routine to search for free entries
    in 'windows'. }

  function which_window( wind_handle: integer ): window_range;

  var
    i: integer; { Used to index through the 'windows' array }
    found: boolean; { Used to indicate when we find a matching window }

  begin
    i := 1;
    found := false;
```



```

while (i <= max_windows) and not found do
  { If we find a matching window, then set our exit flag }
  if windows[i].handle = wind_handle then
    found := true
  else
    i := i + 1;
  if found then
    which_window := i
  else
    which_window := No_Window
  end;
end;

{ draw_object - Draw a single object in a window. The parameter 'obj' is
the record that holds all of the relevant information. We also receive
the two parameters 'x0', and 'y0', which are the coordinates of the
upper left of the window in which we are drawing. Since the coordinates
in 'obj' are "relative" to the origin of the window, we need to add in
these upper left values. }

procedure draw_object( var obj: object_info; x0, y0: integer );
begin
  with obj do
    begin
      Paint_Style( pattern ); { Set the style and color of our object }
      Paint_Color( color );
      { Now we draw the correct object. }
      case shape of
        square: Paint_Rect( x0+x, y0+y, size, size );
        circle: Paint_Oval( x0+x+(size DIV 2), y0+y+(size DIV 2),
                           size DIV 2, size DIV 2 );
      end;
    end;
  end;
end;

```

```
wedge: Paint_Arc( x0+x, y0+y, size, size, 2250, 3150 )
end
end
end;
```

{ draw\_list - Draw the entire object list for a window. Since new objects are added at the end of the 'objects' array, and since objects drawn later should be "on top of" previous objects, we will go forward through the 'objects' array, calling 'draw\_object' for each element. Before we start drawing, we need to determine the upper left corner of the window, so we do a 'Work\_Rect' call, which returns the coordinates of the window workspace area. Notice that this routine does not set the clipping rectangle, since it is designed to be called from the redraw routine. It assumes the proper clipping has already been set up. }

```
procedure draw_list( index: window_range );
```

```
var
  work_x, work_y, work_w, work_h: integer;
  i: obj_range;

begin
  with windows[index] do
    begin
      Work_Rect( handle, work_x, work_y, work_w, work_h );
      for i := 1 to obj_count do
        draw_object( objects[i], work_x, work_y )
      end
    end
  end;
```



{ do\_redraw - Redraw a portion of a window. This routine is called when GEM has sent us a redraw message. The parameters 'x0', 'y0', 'w0', and 'h0' are the rectangular area of the screen which needs to be redrawn. We will intersect that rectangle with the area our window occupies and draw any graphical objects which occupy those areas. The 'do\_redraw' routine doesn't actually put any new data onto the screen itself; it just finds out the index of the window in our 'windows' array and calls 'draw\_list' to draw the window's objects after setting the proper clipping values. }

```
procedure do_redraw( window, x0, y0, w0, h0: integer );
```

```
var
```

```
  x, y, w, h: integer;
```

```
  index: window_range;
```

```
begin
```

```
  index := which_window( window ); { Find out the window's index }
```

```
  { The next code is copied almost verbatim out of the Personal Pascal manual. The only real difference is the call to 'draw_list'. }
```

```
  Hide_Mouse;
```

```
  First_Rect( window, x, y, w, h );
```

```
  while (w <> 0) or (h <> 0) do
```

```
  begin
```

```
    if Rect_Intersect( x0, y0, w0, h0, x, y, w, h ) then
```

```
    begin
```

```
      { At this point, x, y, w, and h hold the coordinates of a rectangle we need to redraw. We first set the clipping to these values, paint the area white, and then call 'draw_list' to redraw any objects that occupy this area. }
```

```
      Set_Clip( x, y, w, h );
```

```
      Paint_Style( Solid );
```

```
    Paint_Color( White );
    Paint_Rect( x, Y, w, h );
    draw_list( index )
end;
    Next_Rect( window, x, Y, w, h ); { Move to the next rectangle }
end;
    Show_Mouse
end;
```

{ add\_object - Add a new object to the list in window 'index' using the currently selected drawing attributes (size, shape, and pattern). The coordinates where the user wanted the object are passed in the variables 'mx' and 'my' (for "mouse x" and "mouse y"). Since we want to store the position of the object "relative" to the upper left corner of the window, we need to subtract the upper left coordinates from these values (the upper left position is added back in whenever the window is redrawn). The drawing attributes are stored as menu item indices, so we have three IF statements to convert to values to store in the 'object\_info' record. The color of the objects just cycles through the range 1 to 3. }

```
procedure add_object( index: window_range; mx, my: integer );
```

```
var
    wx, wy, ww, wh: integer;
begin
    with windows[index] do
        begin
            obj_count := obj_count + 1;
            Work_Rect( handle, wx, wy, ww, wh );
```



```

with objects[obj_count] do
begin
  x := mx-wx;
  y := my-yy;
  { Determine the object's size... }
  if cur_size = size_small then size := 15
  else if cur_size = size_medium then size := 45
  else size := 70;
  { and the object's shape... }

  if cur_shape = shape_square then shape := square
  else if cur_shape = shape_circle then shape := circle
  else shape := wedge;
  { the fill pattern as well... }
  if cur_pattern = pattern_solid then pattern := 1
  else if cur_pattern = pattern_checked then pattern := 23
  else pattern := 0;
  { and finally set the color (we just cycle through colors 1 to
  3 in order to get a pleasing result). }
  color := (obj_count MOD 3) + 1
end;
{ Now we need to draw the new object onto the screen. We first set
the clipping rectangle to our window's workspace area, and then
just call 'draw_object'. }
Set_Clip( wx, yy, ww, wh );
Hide_Mouse;
draw_object( objects[obj_count], wx, yy );
Show_Mouse
end
end;

```

```
{ do_close - Close a window specified by 'wind_handle'. We need to find the
index of the window in our 'windows' array, ask GEM to close and delete
the window, and then remove the window from our records, also. }
```

```
procedure do_close( wind_handle: integer );
```

```
var
  index: window_range;
```

```
begin
```

```
  index := which_window( wind_handle );
```

```
  Close_Window( wind_handle );
```

```
  Delete_Window( wind_handle );
```

```
  window_count := window_count - 1;
```

```
  windows[index].handle := No_Window;
```

```
end;
```

```
{ do_open - Try to open a new window with no objects. We can successfully
create a new window only if both 1) We haven't already opened the maximum
number of windows, and 2) GEM returns us a valid window handle. If we
are successful, we open the window to the maximum size available. Note
that no drawing operations are performed in this routine! The first
event we will receive after opening a new window will be a request from
GEM to redraw our new window, so we don't need to do anything here. }
```

```
procedure do_open;
```

```
var
```



```

junk: integer;
new_index: window_range;

begin
  if window_count = max_windows then
    junk := Do_Alert( '[1][There are no more windows available][ OK ]', 1 )
  else
    begin
      new_index := which_window( No_Window );
      with windows[ new_index ] do
        begin
          handle := New_Window( G_Name|G_Size|G_Close|G_Move, name,
                                0, 0, 0, 0 );
          if handle = No_Window then
            junk := Do_Alert( '[3][GEM is out of windows!][ OK ]', 1 )
          else
            begin
              obj_count := 0;
              Open_Window( handle, 0, 0, 0, 0 );
              window_count := window_count + 1;
            end
          end
        end
      end
    end
  end;

  { adjust_menu - Put checkmarks beside the menu items corresponding to the
    current drawing attributes. }

  procedure adjust_menu;

```

```
begin
  Menu_Check( menu, cur_size, true );
  Menu_Check( menu, cur_shape, true );
  Menu_Check( menu, cur_pattern, true )
end;

{ do_menu - Perform a menu option. Except for the desk title, all of the
  menus just change the current attributes for drawing objects (size,
  shape, etc.), so we'll just erase the checkmark by the old attribute,
  and remember the new attribute. At the end of the routine, we call
  'adjust menu' to put a checkmark beside the new attribute. If the
  menu selected is the desk title, we call the routine 'do_open' to try
  opening a new window. }
```

```
procedure do_menu( title, item: integer );
begin
  if title = size_title then
    begin
      Menu_Check( menu, cur_size, false );
      cur_size := item
    end
  else if title = shape_title then
    begin
      Menu_Check( menu, cur_shape, false );
      cur_shape := item
    end
  else if title = pattern_title then
    begin
      Menu_Check( menu, cur_pattern, false );
```



```

cur_pattern := item
end
else { title must be the desk title }
do_open;
adjust_menu;
Menu_Normal( menu, title )
end;

{ do_message - Handle a message event from GEM. }

procedure do_message( msg: Message_Buffer );

begin
case msg[0] of
  MN_Selected: do_menu( msg[3], msg[4] );
  WM_Topped:   Bring_To_Front( msg[3] );
  WM_Redraw:   do_redraw( msg[3], msg[4], msg[5], msg[6], msg[7] );
  WM_Sized,
  WM_Moved:    Set_WSize( msg[3], msg[4], msg[5], msg[6], msg[7] );
  WM_Closed:   do_close( msg[3] )
end
end;

{ in_window - Test to see if a point is within the working area of a window.
Return true, if the point is within the workspace, and false otherwise. }

function in_window( px, py, wind_handle: integer ): boolean;

```

```
var
  x, y, w, h: integer;

begin
  Work_Rect( wind_handle, x, y, w, h );
  in_window := (px >= x) and (py >= y)
    and (px < x+w) and (py < y+h)
end;

{ do_button - Handle a mouse button event. This routine is called when the
main event loop gets a "mouse button down" event. The position of the
mouse when that event occurred is passed in the parameters 'mouse_x' and
'mouse_y'. We first need to wait for the mouse button to be released,
then check to see that 1) the mouse position is in the top window, and
2) the top window is one of our windows. If these two conditions are
true, and the window doesn't already contain the maximum number of
objects, we call 'add_object' to insert the new object. If we already
have the maximum number of objects, we just inform the user with an
alert box. }
```

```
procedure do_button( mouse_x, mouse_y: integer );

var
  junk, index: integer;
  msg: Message_Buffer;

begin
  junk := Get_Event( E_Button, 1, 0, 1, 0,
    false, 0, 0, 0, 0, false, 0, 0, 0, 0,
    msg, junk, junk, junk, junk, junk );
```



```

index := which window(Front_Window);
if in_window( mouse_x, mouse_y, Front_Window )
and (index <> No_Window) then
  if windows[index].obj_count < max_objects then
    add_object( index, mouse_x, mouse_y )
  else
    junk := Do_Alert(
      ,[1][You've already added the maximum number of objects!][ OK ],1)
end;

```

{ event\_loop - The "heart" of the program. Here we loop forever, waiting for user events, and go to subordinate routines to process them. If we just got a "close window" message, and closing the window left us with no open windows, then we are finished and we return to our caller. Note that the event processing is bracketed by 'Begin\_Update' and 'End\_Update'. The first call disables the menu at the top of the screen so we won't be interfered with while processing our event. }

```

procedure event_loop;

var
  event, junk, mouse_x, mouse_y: integer;
  msg: Message_Buffer;

begin
  repeat
    event := Get_Event( E_Message|E_Button, 1, 1, 1, 0,
      false, 0, 0, 0, 0, false, 0, 0, 0, 0, 0,
      msg, junk, junk, junk, mouse_x, mouse_y, junk );
    Begin_Update; { Make sure no new menu selections occur }
  until

```

```
if event&E_Message <> 0 then
  do_message( msg )
else
  do_button( mouse_x, mouse_y );
End_Update { Allow menu selections again }
until (event&E_Message <> 0) and (msg[0] = WM_Closed)
and (window_count = 0);
end;

{ make_menu - Create a new menu and add all of our titles and items. The
  pointer to the newly created menu is stored in the variable 'menu'. }

procedure make_menu;

begin
  { We need a menu with room for 12 titles and items }
  menu := New_Menu( 12, ' New window!' );

  { Set up window titles }
  size_title := Add_MTitle( menu, ' Sizes ' );
  shape_title := Add_MTitle( menu, ' Shapes ' );
  pattern_title := Add_MTitle( menu, ' Patterns ' );

  { Then the individual items }
  size_small := Add_MItem( menu, size_title, ' Small ' );
  size_medium := Add_MItem( menu, size_title, ' Medium ' );
  size_large := Add_MItem( menu, size_title, ' Large ' );

  shape_square := Add_MItem( menu, shape_title, ' Square ' );
```



```

shape_circle := Add_MItem( menu, shape_title, ' Circle ' );
shape_wedge  := Add_MItem( menu, shape_title, ' Wedge  ' );

pattern_solid := Add_MItem( menu, pattern_title, ' Solid  ' );
pattern_checked := Add_MItem( menu, pattern_title, ' Checkered ' );
pattern_open   := Add_MItem( menu, pattern_title, ' Open   ' );

{ Set the indices of our initial drawing attributes }
cur_size := size_small;
cur_shape := shape_square;
cur_pattern := pattern_solid
end;

{ Finally! The main program code. We just want to initialize GEM, then
initialize ourselves. We draw the menu, and go to 'event_loop' for the
duration of the program. When it returns, we just clean up and finish. }

begin
  if Init_Gem >= 0 then
    begin
      init_windows;
      make_menu;
      Draw_Menu( menu );
      adjust_menu;
      Set_Mouse( M_Arrow );
      event_loop;
      Erase_Menu( menu );
      Exit_Gem
    end
  end.

```

## APPENDIX A

# A Beginner's Guide to Typing In Programs

---

A computer cannot perform any task by itself. Like a car without gas, a computer has potential, but without a program, it isn't going anywhere. Most of the programs in this book are written in a computer language called BASIC.

### BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, and number is significant. Common mistakes are substituting the letter *O* for the numeral 0, a lowercase *l* for the numeral 1, or an uppercase *B* for the numeral 8. Also, you must be sure to enter all punctuation marks, such as colons and commas, just as they appear in the book. Spacing also can be important. To be safe, type in the listings *exactly* as they appear.

### DATA Statements

Some programs contain a section, or sections, of DATA statements. These lines provide information needed by the program. They are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine may lock up, or crash. The keyboard may seem dead, and the screen may go blank. But don't panic. No damage has been done. To regain control, turn off your computer and then turn it back on. This will erase whatever program was in memory, *so always save a copy of your program before you run it*. If your computer crashes, you can load the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is run. The error message may refer to the program line that READS the data. *However, the error is still in the DATA statements.*

### Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use



to store and retrieve programs from tape or disk. Save a copy of your program so that you won't have to type it in every time you want to use it. Learn to use your machine's editing functions. How do you change a line if you make a mistake? You can always retype the line, but you should at least know how to backspace.

If you're working in the Edit window, you can select Help Edit from the Edit menu to get a list of the single-key editing features built into BASIC.

### Hints for Entering ST BASIC Programs

Here are some tips that will make it easier to enter ST BASIC programs. First, although it may be obvious, it is far easier to enter a program from the Edit window than from the Command window. (To move to the Edit window, type EDIT at the Command window's OK prompt, or choose the Start Edit option from the Edit menu.) The Edit window's full-screen editor is much more convenient for entering program lines than is the Command window's single-line interface. You can also run a program directly from the Edit window (type RUN or choose the Start option from the Run menu). When the program is finished, control returns to the Edit window, so you can immediately modify or add new lines to the program.

The Edit window has one feature that you may or may not appreciate. Until you press Return, the line you're working on will appear in *ghost mode* (the letters will look gray and fuzzy). The purpose of ghost mode is to show which lines you have changed. This is helpful to inexperienced programmers, but, since ghosted letters are harder to read than normal ones, it can be an annoyance. To disable ghost mode, enter this line in the Command window:

**POKE SYSTAB+2,0**

Another way to ease the task of program entry is to increase the speed of the cursor. This is done from the Control Panel. The second slider from the top (the one with a rabbit and a turtle) controls the cursor speed. To increase the speed, click on the slider and drag it to the left (toward the rabbit). To slow it down, drag the slider to the right. You can also turn the keyboard beeping sound off and on by clicking the C key icon in the Control Panel.

## APPENDIX B

# Using the First Book of Atari ST Disk

---

If you prefer not to type in the programs in *COMPUTE!'s First Book of Atari ST*, a companion disk is available that includes all the programs in the book. You can purchase it by calling toll-free 1-800-346-6767 (in New York, 1-212-887-8525). Or you can use the coupon in the back of this book.

The disk that you have purchased from COMPUTE! Publications includes a number of different types of files. On the disk you'll find a menu program written in BASIC that can be used to run any of the BASIC programs on the disk; some data files used by the menu program; all the BASIC programs from the disk; Pascal and C source files; and Pascal and C executable files.

The disk does not contain BASIC or Logo; you'll find these programs on the Language Disk that came with your ST.

### Using the Disk

If you have TOS on disk, insert the TOS disk in drive A. If you have TOS in ROM, insert your Language Disk in drive A. Turn on your ST. Once TOS has loaded, select the proper resolution for the program you want to run (see the specific article) by selecting Set Preferences from the Options menu.

Programs in Chapters 2-5 are written in BASIC (except "Doodler," which is written in Logo). They require that you first load BASIC from your Language Disk. If the Language Disk is not in drive A, insert it. Next, double-click the left mouse button while pointing to the icon for disk A. Find the icon for BASIC.PRG and double-click on it with the left mouse button.

Once BASIC has loaded, turn off buffered graphics: Pull down the Run menu, select Buf Graphics, and click the left mouse button. If you pull down the Run menu again, there should now be no checkmark next to Buf Graphics.

Insert *COMPUTE!'s First Book of Atari ST* disk into drive A, type

**RUN MENU**



and press RETURN. Follow the screen prompts to load the desired program. Anytime you're finished with a program and want to return to the menu, also type RUN MENU. To run Doodler, you'll need to load Logo.

### C and Pascal Programs

Chapters 6 and 7 contain one BASIC program, SIEVE.BAS. The rest of the programs in Chapters 6 and 7 are written in C or Pascal. The disk contains both the source code and the executable code for each program. You can run the executable code from the desktop by double-clicking the left button on the appropriate icon. You can examine or print the source code files from the desktop by double-clicking the left mouse button on the file icon and then selecting Show or Print from the dialog box. In order to modify and compile the source code, however, you'll need an editor and a C compiler for the C programs and *Personal Pascal* from OSS for the Pascal program.

Here is a list of the files in Chapters 6 and 7.

Chapter	Article	Filename	Type of File
6	Introduction to C	SIEVE.BAS	BASIC
6	Introduction to C	SIEVE.PRG	Executable
6	Introduction to C	SIEVE.C	C source code
6	Introduction to C	SIEVE2.PRG	Executable
6	Introduction to C	SIEVE2.C	C source code
6	Moving Objects in C	MFDB.PRG	Executable
6	Moving Objects in C	MFDB.C	C source code
6	Moving Objects in C	MULTIFDB.PRG	Executable
6	Moving Objects in C	MULTIFDB.C	C source code
7	Event Management and Windows in Pascal	WINDDEMO.PRG	Executable
7	Event Management and Windows in Pascal	WINDDEMO.PAS	Pascal source code

# Index

- \*/ (in C programs) 254
- % (in C programs) 256
- = (in C programs) 257
- + (in C programs) 258
- := 287
- ( | ) *See* OR, logical
- ACIA 123
- address variable 124
- ADDRIN 176, 179, 206
- ADDRROUT 176
- AES (Application Environment Services) 26, 28, 133, 135, 167, 175-83, 205
- alert box 33, 298
- animation, *NEOchrome* 220
- animation in C programs 261
- array element 295
- array index 295
- attack (musical) 234
- attributes 170, 172
- attribute values 138
- audio/video port 4
- autobooting 119
- backslash ( \ ) 12, 207
- bank-switching 4
- BASIC 200, 281
  - and C 254-57
  - writing 284
- batch processing 259. *See also* DOS shells
- begin (Pascal keyword) 288
- binary mathematics 266
- BIOS 26, 122
- BLOAD 124
- Boolean math 296
- boot disk 119
- braces (in C programs) 254
- buffer 163
- buffered graphics 37
- button events 300
- buttons 11, 207
- C (language) 133, 168, 176, 249, 281
  - and BASIC 254-57
  - animation 261
  - programming 249-60
  - writing programs 252-60
- CALL 124, 168, 226
- CHARREAD 159
- Chromatic Scale (table) 233
- CIRCLE 188
- CLEARW 188, 204
- clicked ( ) 269
- CLOSE 151, 153
- CLS 204
- color 192-94
- color cycling 220-22
- Command window 142
- compiled language 250, 258, 283
- COMPUTE's First Book of Atari ST* disk, using 325, 326
- constant declaration 285
- CONTRL 137, 140, 169, 170, 175, 178, 195, 205
- Control Panel 22-24, 142
- Control-Z 160
- Converting Flats to Sharps (table) 237
- copy-box tool 219
- copying 14
- crash, computer 172, 179
- cursor 142
- d (in C programs) 256
- decay (musical) 234
- declaration statements 255
- #define 255
- DEFSTR 126, 131
- delimiters 68
- DEPOSIT 226
- desk accessory 22
- dialog box 12, 168, 175, 180, 203, 205-7
- directory pathname 12
- disk directory 8
- disk drives 4
- dollar signs, for strings 126
- DOS 7
- DOS shells 259
- double-clicking 8, 216
- dragging 12-14
- drawing 196-201
- DURATION 239
- duration (musical) 233, 237
- editable text fields 29
- Edit window 142
- ELLIPSE 188
- END 153
- end. (Pascal keyword) 288
- enumerated type 294
- envelope (musical) 234
- EOF 156
- ERASE 265
- error code 173
- errors 157
- EXAMINE 226
- executable program file 9
- external support routines 293
- Fibonacci sequence 40
- FIELD 163
- filename 153
- files 150-66
  - drawers 7
  - mode 162
  - number 153
  - rearranging 30
- FILL 188, 190-92
- fill icon 224
- 1ST Word word processor 5, 122, 166
- FIX-OFFSET 164
- flats (musical) 237
- folder 11, 12
- form (musical) 234, 235
- form\_alert routine 179-80, 206
- FULLW 188
- function 295
- functions and procedures 287. *See also* subprograms
- GB 169, 175
- GEM 5, 27, 135, 204, 251
- GEM desktop 5-25, 27
  - bug 29
  - crash 29
  - customizing 119, 120-22
  - modifying 30
- GEMDOS 26, 27, 135
- GEMSYS 133, 167, 168, 175
- generalized drawing primitive 138
- GET # 163, 164
- getchar ( ) 257
- ghost mode 7, 142, 324
- gintin 134
- GINTIN, GINTOUT 176, 179
- global variable 289
- GOSUB 152
- graphics 187-202, 203
- graphics primitive 170
- Graphics Statements (table) 188
- handle (in Pascal programs) 295
- hard disk interfaces 4
- hot spot, pointer 144
- icons 7, 9-11
- IKBD (intelligent keyboard device) 122
- immediate mode 137
- indentation 254
- index file 151
- initialization sequence 119
- INKEY\$ 204
- INP 190, 204
- INP ( ) 69
- INPUT 69, 151, 153
- INPUT # 151, 158
- INPUT\$ 159
- installing an application 16-19
- Install Printer option 23
- intelligent keyboard device. *See* IKBD
- INTIN, INTOUT 138, 145, 169, 172, 174, 175, 195, 205
- joystick 119, 122-24
- KEY OFF 204
- Language Disk 187, 325
- LBUTTON 146
- left-clicking 215
- line-A routines 27
- line-drawing tool 217
- LINEF 188, 196, 204
- LINE INPUT # 151, 159
- line label 157
- line number 157, 254
- LINEREAD 159, 160
- linker program 259
- LOAD 153
- LOC 163
- LOF 163
- Logo 5, 226
- loop
  - C 256, 269
  - Pascal 285
  - repeat-until 296
- LPRINT 157
- macro 255
- magnify window 218
- main ( ) 254
- mask 147
- Megamax* C compiler 259, 268
- memory addresses 176
- memory form definition block. *See* MFDB
- memory management unit. *See* MMU
- memory register 97
- menu titles 7
- MFDB (Memory Form Definition Block) 261-66
- MMU (Memory Management Unit) 123
- MOD 203
- mode 152
- mode value 265
- monochrome monitor 4
- Motorola 68000 microprocessor 3
- mouse 12, 141-49, 207
  - controller 6, 141
  - customizing 146
  - pointer 180
  - reading 144-46
- msg buffer 296, 297



- multiple-choice test 66
- multiple screen windows 6
- multiple selection 15
- multitasking 23
- music 232-46
- Musical Notes (figure) 237
- NEOchrome program 5, 213-25
  - color chart 215
- NEW 153
- nonexecutable file 9
- Note Duration (table) 239
- numeric values 126
- object code 259
- octave 232, 237
- Octaves (figure) 238
- offset 163
- ON ERROR 156
- opcode 170, 178, 197
- opcode number 136
- OPEN 151
- operations code 197
- OR 266. *See also* XOR
- OR, logical (|) 179, 207
- oscillator (musical) 233
- Output window 133, 204
- palette blocks 218
- parallel port 4
- parameter 254
- parameter blocks 136, 145, 168-70, 175, 205
- parameter list 124
- parentheses 254, 256
- Pascal programming 281-89
  - main body 288
  - programs 285
- PCIRCLE 188
- PELLIPSE 188
- period (musical) 235
- PERIOD 240
- Personal Pascal 32, 291, 295
- pitch (musical) 232, 237
- pixels 214
- pointer 262
- predefined constants 293
- preprocessor 255
- prime numbers 252
- primitive ID 138, 170-72
- PRINT 157
- PRINT # 151, 157
- printf() 256
- procedure, Pascal 295
- program identifier 285
- pseudo-code 250
- PSG (Programmable Sound Generators) 241
- PTSIN, PTSOUT 138, 145, 169, 171, 174, 175, 195, 205
- put() 266
- PUT # 163
- question mark (in Logo programs) 227
- QUIT 153
- RAM disk 259
- ramp lines 217-20
- ramp pointers 218
- random files 151, 160-64
- range (musical) 232
- RBUTTON 146
- READ 154
- READFILE routine 152, 156
- record length 162
- records 163
- redraw 302
- release 234
- remark 254
- RENUM 157
- REPLACE 265
- reserved memory 171, 205
- reserved variables 175
- RESUME 156, 157
- REVerse TRANSPARENT mode 265
- right-clicking 215
- root directory 12
- roping 15
- RUN 142, 153
- Save Pic option 229
- screen modes 19-22
- screens, full 204
- semicolon 254, 256, 285
- sequential files 68, 151-60
- set mouse form. *See* SMF
- Settings menu 226
- shape storage modes, converting 264
- sharps (musical) 237
- SKETCH 227, 228
- SLIDENEO program 221
- sliders 214
- SMF (Set Mouse Form) 146
- sorting routines 126-32
- SOUND 232
- sound generator 241
- source code 258
- source image 264
- sprites 261-69
- sprite simulation 261
- ST BASIC 5, 135, 203, 204, 324
  - file processing 69
- ST Writer word processor 120, 164
- Start option 142
- static palette 221
- STICK 122
- STRIG 122
- string, define 33
- strings 126
  - sorting 131
- structured language 251
- subfunctions 138
- subprograms 287
- subroutines 138
- sustain (musical) 234
- SWAP 126
- SYSTAB 190
- system routines 135-49, 168
- tempo 240
- Tempo (table) 240
- text, writing 194-96
- text editor 120
- timing delay 223
- TOS 5, 26, 135, 203, 282
- TRANSPARENT mode 265
- TRAP 27
- trash can 7, 121
- Turbo Pascal 282
- two-drive system 8
- type 31, 293-94
- type declaration 286
- typing in programs 142, 323, 324
- UNIX operating system 249
- value 294
- variables 81, 126, 255, 285, 293
- VARPTR 124, 178, 193
- VDI 26, 28, 133, 135, 167, 187, 194-202, 204
  - calling the system routine 173
- library 261
- opcodes 136
- Raster Opaque Copy Form 261
- system routine 136
- VDI and AES Routines (table) 208-9
- VDISYS 133, 135-41, 167, 168, 173-75, 178, 205
- vertical synchronization 268
- vertices 137, 170
- voice (musical) 232
- volume (musical) 232
- vro\_cpyfm() 261, 265
- Vsync() 268
- VT-52 terminal emulator 23
- WAVE 232, 233
- Waveforms (figure) 235
- WAVES (table) 234
- windows 291-304
  - active 9
- Wirth, Niklaus 281
- with (Pascal keyword) 295
- WRITE # 151, 157, 158
- XBIOS 26, 27
- XOR 265-67

To order your copy of *COMPUTE!'s First Book of Atari ST Disk*, call our toll-free US order line: 1-800-346-6767 (in NY 212-887-8525) or send your prepaid order to:

*COMPUTE!'s First Book of Atari ST Disk*  
**COMPUTE!** Publications  
P.O. Box 5038  
F.D.R. Station  
New York, NY 10150

All orders must be prepaid (check, charge, or money order). NC residents add 4.5% sales tax. NY residents add 8.25% sales tax.

Send \_\_\_\_\_ copies of *COMPUTE!'s First Book of Atari ST Disk* at \$15.95 per copy. (203BDSK)

Subtotal \$\_\_\_\_\_

Shipping and Handling: \$2.00/disk \$\_\_\_\_\_

Sales tax (if applicable) \$\_\_\_\_\_

Total payment enclosed \$\_\_\_\_\_

☐ Payment enclosed

☐ Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No. \_\_\_\_\_ Exp. Date \_\_\_\_\_  
(Required)

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

Please allow 4-5 weeks for delivery.



To order your copy of COMPACT, call 1-800-832-5252 or send your check to:

COMPACT, Dept. of Agriculture  
P.O. Box 5000  
Ft. Collins, CO 80521  
New York, NY 10000

Amount of the check should be \$10.00 plus \$2.00 shipping and handling fee. Payment should be made by check or money order.

Send this order form with your check to: Dept. of Agriculture, P.O. Box 5000, Ft. Collins, CO 80521.

Name \_\_\_\_\_

Shipping and handling \$2.00 per copy

Address \_\_\_\_\_

Total payment enclosed \_\_\_\_\_

Payment enclosed  
to Charles & Joan Moore, Dept. of Agriculture

Order No. \_\_\_\_\_ Date \_\_\_\_\_

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_

State \_\_\_\_\_ Zip \_\_\_\_\_

# CHARTER SUBSCRIPTION FORM

☐ Payment enclosed ☐ Charge my VISA/MasterCard

☐ **YES!**

Sign me up for six issues (a full year's subscription) at the special introductory price of just \$59.95. I save more than \$17 off the newsstand price.

Credit Card # \_\_\_\_\_ Exp. Date \_\_\_\_\_

Signature \_\_\_\_\_

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

Outside U.S.A., please add \$6 (U.S.) per year for postage.

# CLIP THIS AND SAVE \$17

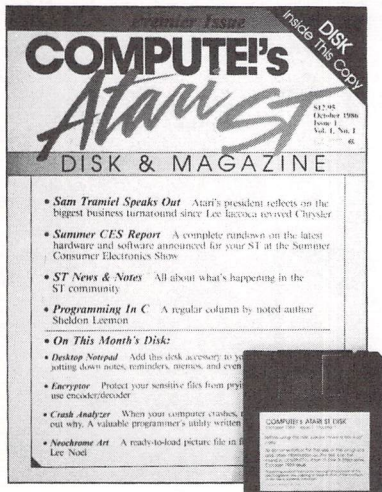
Here's your chance to cash in with big savings on *COMPUTE!'s Atari ST Disk & Magazine*—the exciting new publication devoted exclusively to the special needs and interests of Atari ST users like you.

Every other month, *COMPUTE!'s Atari ST Disk & Magazine* brings you exciting new action-packed programs already on disk! Just load and you're ready to run.

You can depend on getting at least five new programs in each issue—high-quality applications, educational, home finance, utility, and game programs you and the entire family will use, enjoy, and profit from all year long.

And here's even more good news. Subscribe now to *COMPUTE!'s Atari ST Disk & Magazine* and take advantage of big Charter Subscription savings. Get a full year's subscription for just \$59.95. You save over \$17 off the newsstand price.

No other publication gives you more for your Atari ST than *COMPUTE!'s Atari ST Disk & Magazine*. So sign up now by using the coupon above—or call 1-800-247-5470 (in Iowa 1-800-532-1272).





NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

## BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 7551 DES MOINES, IA

POSTAGE WILL BE PAID BY ADDRESSEE

### COMPUTE!'s Atari ST Disk & Magazine

P.O. Box 10775

Des Moines, IA 50347-0775



NEW FOR ATARI ST USERS

# COMPUTE!'s ATARI ST DISK & MAGAZINE

**Only *COMPUTE!'s Atari ST Disk & Magazine* gives you all this and more in each big issue:**

**TOP QUALITY PROGRAMS:** Application programs for home and business. Utilities. Games. Educational programs for the youngsters. All are already on an enclosed disk and ready to run. For example: a typical disk might contain an elaborate adventure game written in BASIC, a programming utility written in machine language, a dazzling graphics demo in compiled Pascal, and a useful home or business application written in Forth or C.

**NEOCHROME OF THE MONTH:** What are computer artists doing with the Atari ST? Each issue contains a Neochrome picture file—ready to load and admire.

**REGULAR COLUMNS:** If you're a programmer—or would like to be—you'll love our col-

umns on ST programming techniques and the C language. Or check out our column on the latest events and happenings throughout the ST community. Or send your questions and helpful hints to our Reader's Feedback column.

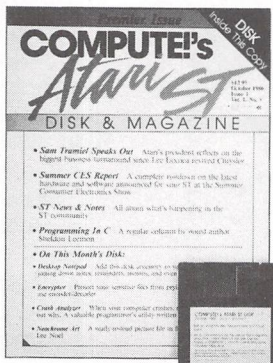
**REVIEWS:** Honest evaluations of the latest, best software and hardware for the Atari ST.

**NEWS & PRODUCTS:** A comprehensive listing of all the new software and peripherals for your ST.

**AND MORE:** Interviews with ST newsmakers, reports on the latest industry trade shows, and overviews of significant new product introductions.

Don't miss a single big issue. Subscribe to *COMPUTE!'s Atari ST Disk & Magazine* now through this special money-saving offer. Return coupon above or call 1-800-247-5470 (in Iowa 1-800-532-1272).

**COMPUTE! Publications, Inc.**  
Part of ABC Consumer Magazines, Inc.  
One of the ABC Publishing Companies



RETURN COUPON ABOVE TO ENJOY  
CHARTER SUBSCRIPTION PRIVILEGES





# An ST Anthology

This first COMPUTE! collection of games, applications, and programming for the Atari 520ST and 1040ST personal computers includes material sure to fascinate every ST user. There are over 25 articles, many never before published, from games to explorations of ST sound and graphics capabilities to information, utilities, and tutorials for programmers. Here's just a preview of what's in store:

- "Home Financial Calculator," a program that combines ease of use and versatility, and integrates a variety of loan and investment features
- "Reversi," an adaptation of a classic strategy game
- "Multiple-Choice Test Generator," a learning aid that you can use to make up your own tests
- An introduction to programming in C and Pascal
- Access to system routines through GEMSYS and VDISYS
- A guide to creating *animated* figures with NEOchrome
- A tutorial on designing custom title bars
- A music-generating program that enables you to add music to your own programs
- And much, much more

Practical, enjoyable, and understandable, *COMPUTE!'s First Book of Atari ST* is clearly written and will help you get the most out of your computer. Like all COMPUTE! books, this one is designed to solve some of the mysteries that slow your progress to fuller comprehension of your computer's features. Each program has been thoroughly tested and is ready for you to type in and run. Or, if you prefer not to type in the programs, you can order a disk that contains all the programs in the book. There's a wealth of information here for every ST user.

*The programs in this book are available on a companion disk. See the coupon in the back for details.*